

HotString Helper 2.0 and [AutoCorrect for AHK v2](#)

kunkel321 | User manual updated 11-6-2024

Welcome

My hope is that an overview of hotstrings will help explain why HotString Helper 2.0 and the AutoCorrect f() Function, as well as the many word part entries, are made the way they are. Whether good or bad; this manual reads more like a series of journal entries than a traditional software user manual. My advance apologies for the topic jumping and the tangents. I recommend reading the sections in order, since they build on each other.

Table of Contents

Welcome	1
Brief Overview of HotStrings	2
Uses of HotStrings	2
Making a Boilerplate Text Expansion.....	3
AutoCorrect	3
Philosophy of best multi-word matches.....	4
Original HotString Helper	6
HotString Helper 2.0 (hh2).....	6
Boilerplate Entries	6
Following Tabs	8
Show Symbols.....	8
Make Replacement Box Bigger.....	9
AutoCorrect word entries.....	10
Auto Entry of New AC Items	10
Word Analysis.....	11
Delta String	11
Trimming	12
Add-a-Letter	12
Check For, and Capture, Existing HotString	12
The Comparison Word List	13
The Row of Six Buttons	13

Validity Checks.....	14
Conflict Scenarios	16
Special situations with word-ending multi-match autocorrect entries	17
When misspellings are tolerated	17
Using Case Sensitivity to protect words	18
Context Sensitive Hotstrings.....	18
Inevitable Errors	19
Regarding keyboard input buffering.....	19
Regarding confounding mis-typings	19
AutoCorrection Logging.....	20
The AutoCorrectsLog File.....	20
AC Log Analysis	21
HotString Logging Circular Logic Problem.....	22
HotString Rarefication	22
The f() Function	23
AutoCorrect Fixes Report	23
Some Caveats	24
The CAsE COrrector tool.....	25
Accented words with diacritic characters.....	25
Date Tool - H.....	26
Manual Correction Logger (MCLogger)	27
Removed HotStrings List.....	29
AutoCorrect Script Updater Tool	29
Conflicting or Nullified HotStrings Finder Tool	30
Color Theme Integrator Tool	31
End.....	33

Brief Overview of HotStrings

A “hot string” is a predefined sequence of keys that, when pressed in order, will trigger some action by the software. For the purposes of this discussion, we are referring to *auto-replace* hotstrings, where the triggering characters are automatically removed. A hotstring mustn’t be confused with a “hotkey combination,” where multiple keys are pressed at once, simultaneously. With hotstrings, the software—in this case, AutoHotkey—watches for the string. When the series of keys are typed by a user, the software detects the input sequence, then simulates another, associated, series of key presses which are the pre-defined expansion, or “replacement” text. AutoHotkey is unique in the PC world, partly because of its ease of use, with regard to creating custom hotstrings. Indeed, the desire to allow hotstrings to be constructed in a single line of code was a big part of the impetus for the original creation of AutoHotkey (AHK). For interested readers who are new to hotstrings, a good place to start is the AHK [HotStrings Documentation](#). Intermediate AHK users will know that there is a [Hotstring Function](#) that programmatically creates hotstrings at runtime. The below discussion covers the first type of hotstrings, not the use of the Hotstring Function.

Uses of HotStrings

In AHK, a hotstring might be used to activate a block of code, but a more-frequent use is to auto-replace text. “Auto-replacement” hotstrings typically have at least two components: **Trigger** and **Replacement**. An example from the AHK docs is:

```
::btw::by the way
```

The AutoHotkey application knows that this is a hotstring definition because of the two sets of double-colons. While running this code, type “btw” and upon pressing an “End Character” such as space, period, enter, etc, it will be replaced with “by the way.” To add more utility, AutoHotkey allows various options that affect how the hotstring works. The options must appear between the first set of colons. Additionally, AutoHotkey allows *in-line comments* to be added at the end of individual lines of code. In-line comments always begin with a space, then semicolon, and are ignored by the AHK application. They are only there for human consumption. Here is another example:

```
::*:fwiw::for what it's worth ; the asterisk means that no end char is needed.
```

Making a Boilerplate Text Expansion

When AHK users create their own hotstrings, they usually create boilerplate “template” items. Examples include the two above (btw and fwiw). Another example is:

```
:::sig/::Stephen Kunkel`nAutoHotkey Enthusiast`n555-1234
```

Upon typing “sig/” the instant the slash is pressed, the software would replace the trigger with:

```
Stephen Kunkel  
AutoHotkey Enthusiast  
555-1234
```

Please note that the trigger string is suffixed with a slash. A suffix is not the same thing as an End Char. Indeed, no end char is needed here, because of the asterisk option. It is important for the suffix “/” to be there though, otherwise you couldn’t type words beginning in “sig” because it would trigger an unwanted replacement. Having a trigger string prefix would also work (e.g. /sig, .sig, or ;sig). Rather than using part of a word as the trigger, a user might also choose an *acronym* (technically an “*initialism*”) made from the first letter of each word (skae).

Another relevant feature is that multi-line expansions like the one above can be made via *Continuation Section* such as:

```
::skae::  
(  
Stephen Kunkel  
AutoHotkey Enthusiast  
555-1234  
)
```

This adds more lines to your script file, but it has the effect of being more “*what you see is what you get*,” which is nice. As we will see below, HotString Helper 2.0 makes multiline hotstrings, by default, using a Continuation Section. It is super important to remember that, if you bulk sort your hotstrings, DO NOT sort the Continuation Section-style hotstrings.

AutoCorrect

Another common use of AutoHotkey auto-replace hotstrings is to have personal *autocorrect libraries*. Long-time users of Microsoft Word will be familiar with Word’s AutoCorrect feature which has a library of common typos and misspellings. When you type one of them, Word instantly fixes the error. This is not the same as *SpellCheck*, which locates your misspellings and suggests corrections. AutoCorrect only works *as you type* and it makes changes without prompting the user. An AHK script file with a library of autocorrect items works essentially the same way—but it is not limited to MS Office applications. It works most anywhere that you type text on your PC.

A fundamental difference between boilerplate hotstrings and autocorrect hotstrings is that you *intentionally* place boilerplate items in your documents, emails, etc. You don’t intentionally misspell or mistype words or phrases. AutoCorrect items are preemptively added to your system as a safety net, or failsafe.

Many AHK users will already be familiar with the most excellent [2007 AutoCorrect.ahk](#) script by Jim Biancolo. The list, that is written in AutoHotkey v1 code, was largely derived from Wikipedia’s list of common misspellings. An updated, for [AHK v2, version of AutoCorrect.ahk](#), by the current writer (kunkel321) added about 1200 “[common grammar errors](#),” also from Wikipedia.

Philosophy of best multi-word matches

What makes a good autocorrect item? Certainly, it makes sense to have corrections for words that are *high-frequency words* in the given language; English for our purposes here. Secondly, having corrections for likely-to-occur misspellings or mis-typings makes sense. As mentioned above, the ubiquitous 2007 AutoCorrect.ahk was based primarily on [Wikipedia's lists of common errors](#). But where does that list come from? Of course, it's the [Wikipedia Typo Team](#)! All in all, they do a fantastic job, but let's face it: With 6.7 million articles, written using 4.3 billion words, the Typo Team can't be physically proofreading all of those. My own hypothesis is that different versions of Wikipedia are programmatically compared, and when a particular word is replaced with another similar particular word, X number of times, the different versions of the word are saved and flagged as "*frequently occurring*" typo and correction pairs. If that is how they do it, it seems like a solid approach, but I think that sometimes a person writes some really long article on an obscure topic, then someone else comes along and changes the spelling of some word that occurs X times, but *only occurs in said article*. That might explain some of the seemingly obscure typo-correction pairs in the list, such as: *alsation->Alsatian*, *bernouilli->Bernoulli*, *lotharingen->Lothringen*, *lukid->likud*, *mythraic->Mithraic*, *papanicalou->Papanicolaou*, and *valetta->Valletta*.

Whatever the reason for the odd entries in AutoCorrect 2007, most of the words are good useful words. It is noteworthy that the list contains 4637 whole-word corrections. Many of those words have common root words but differ in prefixes (such as "*un-*" or "*anti-*" or "*re-*") or suffixes (such as "*-ly*" or "*-ing*" or "*-ness*"). When working with simple basic AHK hotstrings (those without hotstring options), it is indeed necessary to use whole words. And so, including the various permutations (single/plural, etc) for these items is needed. For example, consider this item:

```
::mrak::mark
```

Typing "mrak" results in "mark." But typing "unmrak" does not result in "unmark," because by default, hotstring triggers can't be preceded by alphanumeric characters. Similarly, "mraked" doesn't get corrected, because an End Char needs to be typed to trigger the replacement. There are however, [hotstring options](#) for both scenarios.

```
:?:mrak::mark ; This word-end item corrects mrak and also unmrak.
```

```
:*:mrak::mark ; This word-beginning item corrects mrak and also mraked.
```

They can be combined as well.

```
:*?:mrak::mark
```

This is a "word middle" match and fixes "birthmark," "Denmark," "earmarked," and 172 other words. Of course, not all of the 175 potential fixes are high-frequency words. I mean, how often will a person type the word "supermarketeer?" Still, 175 potential matches for a single autocorrect entry is pretty impressive, in terms of being high-utility.

Jim recognized this potential when making the original 2007 AutoCorrect.ahk script. The script contains 83 word beginning items, that use the `:*` option, 15 word ending items that use `:?:`, and two word middles with both options `:*?:`.

```
:*?:comptab::compatib ; Covers incompat* and compat*
```

```
:*?:catagor::categor ; Covers subcategories and categories.
```

I had never really expanded on this concept, until being inspired by Jack Dunning's (AHK v1) book, [Beginning AutoHotkey Hotstrings: A Practical Guide for Creative AutoCorrection, Text Expansion and Text Replacement](#). In chapter three he introduces the reader to using a site that looks up words via wildcard matches, to see how many matches there are to various word parts. For example, referring to these two items:

```
:*?:ugth::ught
```

```
:*?:igth::ight
```

Jack writes, "According to the word lookup site referenced above, the first line of code protects against the misspelling of 133 words through the accidental swapping of the ht. The second line adds another 887 word variations to the same AutoCorrect app." As pointed out by Jack (and as logically concluded), the key here; the way to make the best multi-match high-utility autocorrect items, is to identify likely misspellings that correspond to a large number of words, but that also don't inadvertently misspell other words.

In 2021 I began the process of manually assessing and converting all 4637 whole-word hotstrings, from the 2007 AutoCorrect list. I already had the excellent application, [WordWebPro](#) installed. WordWeb has a utility for doing wildcard searches, so I used that tool, manually pasting in each word, then progressively deleting and backspacing letter-by-letter from the beginnings, then the ends of

each word, to see which word middles were good multi-match items. An important part of the process was to also trim off the same letters from the beginning/ending of the (misspelled/typo) hotstring trigger string, to ensure that the new, shortened, trigger didn't correspond to—and therefore inadvertently misspell—any other words. This was a mind-numbingly tedious and time-consuming process. I eventually became sufficiently motivated to create the [Word Analyzer Gui \(WAG\) Thing](#). As indicated in the WAG forum post,

“The WAG tool won’t convert the hotstrings for you, but it will let you simultaneously winnow-down the trigger string and the replacement string and compare each to a list of English words. The winnowing process usually involves trimming the prefix or suffix from a root word (assuming that the typo is in the root). It doesn’t “intelligently” remove the same letter from each string, it just removes a letter-at-a-time”.

The resulting multi-match autocorrect entries—at least the good ones—are *high-utility items*. The WAG tool itself, however, is not a *high-utility tool*. Really, its only purpose for existence is to process autocorrect items. It is a “one-trick pony” in that regard. It served its purpose well, allowing me to process all the remaining whole word items from the AutoCorrect 2007 list, perhaps 10 or 20 times faster than without it. After completing the list, for a short time, the WAG Thing fell into disuse, but as I later found more whole-word autocorrect items via internet search, ChatGPT, and personal poor typing experiences (then later [the MCL tool](#)), I found that the WAG tool was still useful from time to time. Often, when typing, I’ll misspell or mistype a word, then save it as an autocorrect entry. Then later, when I have time, I’ll go back to and examine the entry to see if it is a good candidate for a multi-match word part entry.

As discussed below, I integrated the tool as a *sub-component of HotString Helper*, and got rid of the “WAG” name, because.... Well, because it’s rather silly to name something “Wag” unless it has to do with happy dogs. I guess we’ll call it the *Exam Pane*.

As an example of how the “word-analyzing/exam” process is occasionally useful, while writing this article—in the first paragraph—I used the word “combination.” Being a person who is bad at spelling, I typed it “combonation.” I can imagine myself potentially misspelling that word the same way in the future, so I decided it is probably a good candidate for an autocorrect item. The item, written for AHK (v1 or v2) would be:

```
:: combonation::combination
```

By simply adding the options, “.*?:”

```
.*?: combonation::combination
```

This becomes a multi-match item, and matches, “combination, combinational, combinations, recombination, and recombinations.”

If we trim five characters off the end, we get,

```
.*?: combon::combin
```

which potentially fixes 40 different words, according to the word list I was using at the time. Note that you can’t remove any letters from the beginning,

```
.*?: ombon::ombin
```

or it will inadvertently change “trombone” to the erroneous “trombine.” So that’s no good.

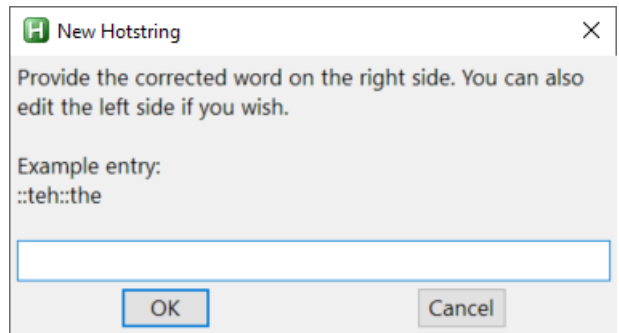
Let’s take a look at the HotString Helper tool, including this “combonation” example, then we’ll talk more about the f() Function and the AutoCorrect for v2 script, as well as a few other tools that were later added to the ac2 package.

Original HotString Helper

On [his blog](#), Jim B. credits Tara Gibb with the original HotString Helper tool, though the [AHK Documentation suggests](#) it was Andreas Borutta. While AutoCorrect.ahk 2007 is running, press the Win+H hotkey, and you'll get the image to the right.

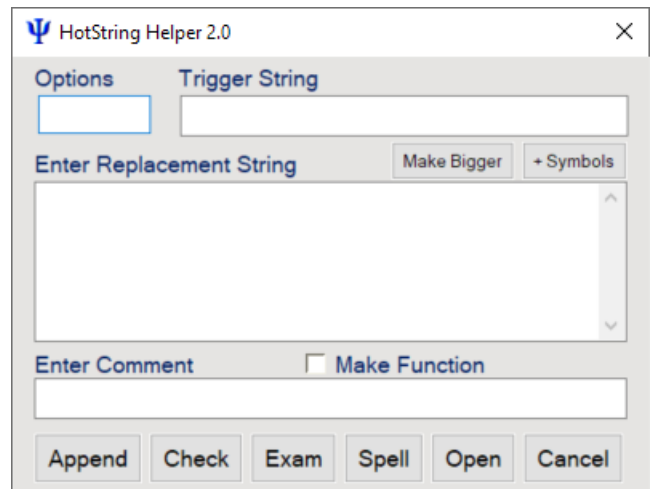
Enter your hotstring and press OK, and the new hotstring gets added to your HotStringLib.ahk file, then the file is saved and reloaded so that your new hotstring is loaded into your PC's RAM and is ready to use. Nice. The version of HotString Helper from 2007 is written in AHK v1, but a [v2 remake](#) is available as well.

It's noteworthy that AHK has a "HotString" function which makes a hotstring immediately available to a script. Though the above-linked v2 remake utilizes the Hotstring *function*, neither hh v1, nor kunkel321's "ac2" uses the function. The ::trigger::replacement style hotstrings only become available when the containing script is reloaded.



HotString Helper 2.0 (hh2)

My own version of HotString Helper works similarly but has several extra features. It's noteworthy that the "2.0" *nomenclature* is not really a version number, per se. It's mostly there to differentiate this version from the 2007 original. I chose "2.0" because it is written in AutoHotkey v2. The original motivation for this version was to facilitate the creation of multi-line Continuation Section boilerplate items, *in addition to* being able to create normal single word autocorrect entries. The first version of this was written in AHK v1 and was called "HotString Helper – Multi-Line." HH-ML was then remade with AHK v2 code. Upon combining the above-mentioned WAG Thing components, I considered, "HotString Helper – Word-Analyzing Multi-Line," but that is too long, and "WAML" is too confusing. So "2.0" it is; or "hh2" for brevity. It appears as in the screenshot on the right. Define your own GUI form colors and icon near the top of the code.



Though I don't really use proper version numbering, there is a version date in the comments near the top of the code.

The first obvious difference with HotString Helper 2.0 (hh2), is that each component of the hotstring has its own edit box.

- [Options](#) (which are optional)
- Trigger String (a.k.a. hotstring; mandatory)
- Replacement String (a.k.a. expansion text, boilerplate text; mandatory)
- Comment (optional)

Boilerplate Entries

My fake signature here is from the above example.

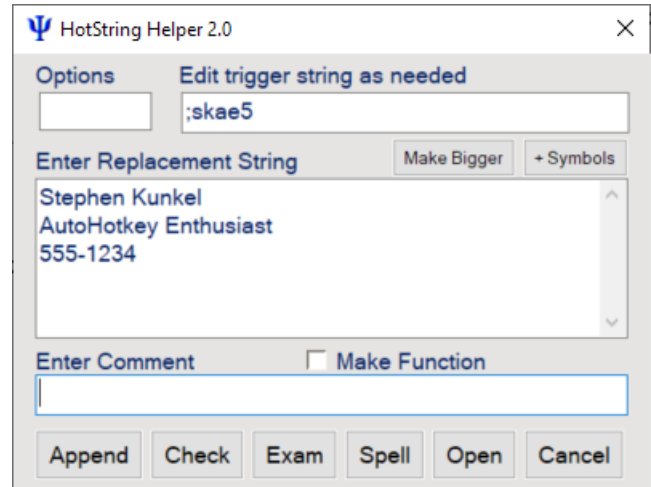
Stephen Kunkel

AutoHotkey Enthusiast

555-1234

If I select the three lines of text, then press the default hotkey: **Win+H** (technically, it's "Win+h"), I get what is shown in the image to the right.

Notice the automatically-generated trigger string, **“;skae5.”** This is an acronym (technically an “initialism”) made from the first letter of each word. It’s prefixed with a semicolon because I always start my boilerplate triggers with a semicolon. The semicolon potentially confuses things, because semicolons delineate in-line comments in AHK code. They’re just so convenient though... Semicolon is one of the eight Home Keys! If you have a different preferred prefix or suffix, define it with the **myPrefix** variable – see below. When creating the acronym, hh2 is coded to use the first character, of the first X words, and to skip words that are Y letters or shorter. Near the top of the code are the variables where X and Y are defined. They are shown below as well.



;===Change=options=for=MULTI=word=entry=options=and=trigger=strings=as=desired==

; These are the defaults for "acronym" based boiler plate template trigger strings.

DefaultBoilerPlateOpts := "" ; PreEnter these multi-word hotstring options; "*" = end char not needed, etc.

myPrefix := ";" ; Optional character that you want suggested at the beginning of each hotstring.

addFirstLetters := 5 ; Add first letter of this many words. (5 recommended; 0 = don't use feature.)

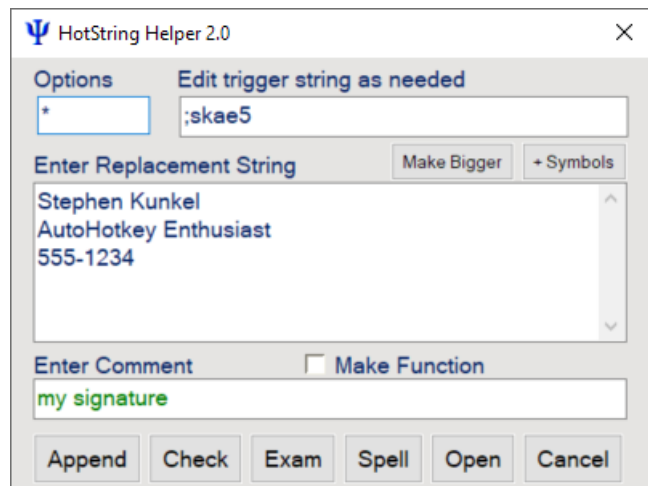
tooSmallLen := 2 ; Only first letters from words longer than this. (Moot if addFirstLetters = 0)

mySuffix := "" ; An empty string "" means don't use feature.

For the purposes of this demonstration, I manually added an asterisk to the Options edit box. This causes the hotstring to activate without the need of an End Char. Therefore, the moment I press the '5' it will trigger the replacement text. I also added a comment, again for demonstration purposes. The image to the right shows the hh2 form with these things added.

Upon clicking the Append button, the following is added to the bottom of the AutoCorrect.ahk file:

```
;*;skae5:: ; my signature
(
Stephen Kunkel
AutoHotkey Enthusiast
555-1234
)
```

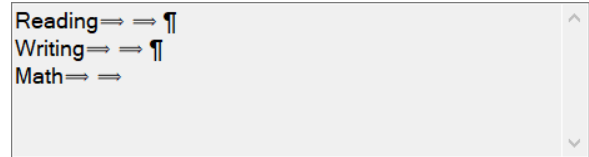


Notes:

- Pressing the *Enter* key will also append the item. However, pressing the Enter key will not have this effect if you are editing the replacement text. When editing the replacement text, the Enter key merely does its default behavior of typing a new line.
- Note also, that holding the *Shift* Key while clicking the Append button (i.e. *Shift+Click*) will save the hotstring to the Windows clipboard instead of appending it to the AutoCorrect file. In this situation, if the Validation Warning appears, you have to keep holding the Shift key for the item to be placed on the clipboard. Note also, the item will only be on the clipboard while the hh2 form is shown. Once you close the form, then the previous Windows clipboard contents will be restored.

Following Tabs

Sometimes I like to have Tab characters to the right of the multiline items, for example these academic test areas in the screenshot:



This allows me to easily add (for example) the reading, writing, and math test scores into their own column to the right of the names, after expanding the boilerplate text. When you select the multiline item, and press Win+H, *if a Tab character is detected at the end of the replacement string, then the command to keep the white space is added to the hotstring, such as:*

```
:::rwm::  
(RTrim0  
reading  
writing  
math  
)
```

Without the *RTrim0* (Right Trim Zero, where zero means 'off'), AHK would remove the last Tab characters when it expands the text. (The tabs are there, they are just invisible.)

Show Symbols

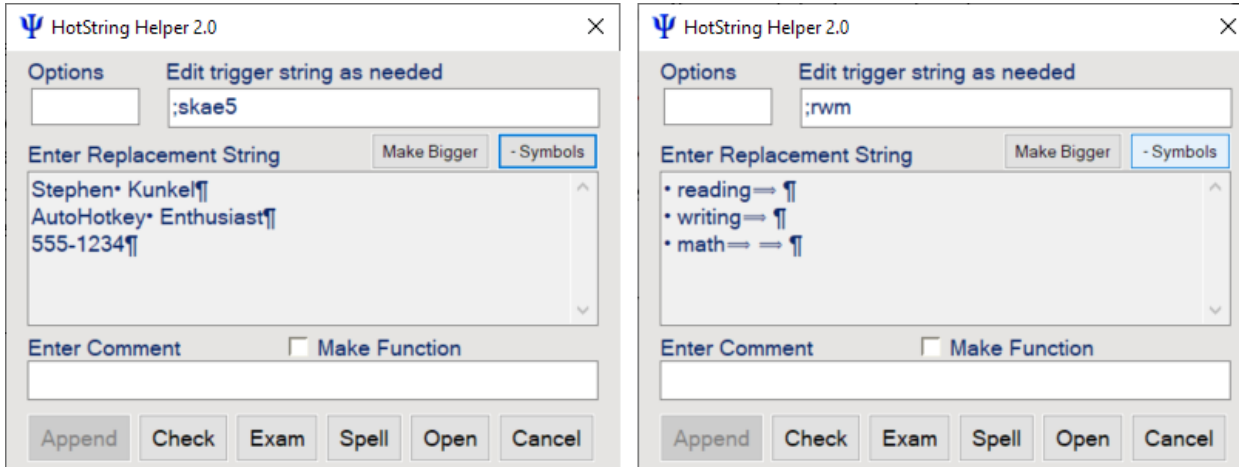
Please note the two small buttons just above the Replacement String edit box. The **+/- Symbols** button toggles the visibility of a *Pilcrow*, *dot*, and *right arrow* (for Enter, Space, Tab, respectively) as in the images below. When the symbols are visible, the Replacement String box becomes *Read-Only*, and the Append button becomes inactive. The actual symbols for these can be changed near the top of the code. The settings are:

```
;====Assign=symbols=for="Show Symb"=button=====
```

myPilcrow := "¶" ; Okay to change symbols if desired.

myDot := "• " ; adding a space (optional) allows more natural wrapping.

myTab := "⇒ " ; adding a space (optional) allows more natural wrapping.



Be sure to have your .ahk file formatted as [UTF-8 with BOM](#) or the symbols might not get displayed correctly.

Special thanks to user "Off" who helped me work out how to create this effect.

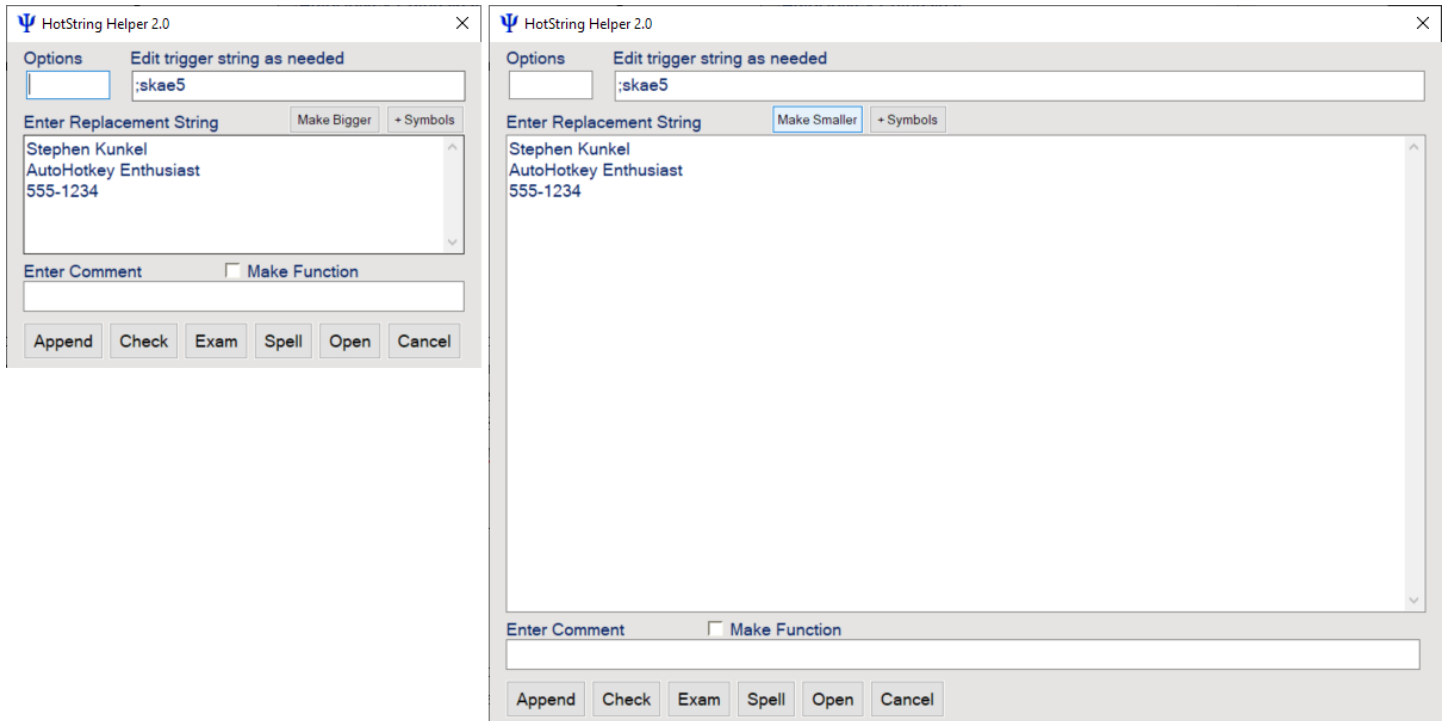
Make Replacement Box Bigger

The **Make Bigger/Smaller** button toggles the size of the Replacement String box, as seen in the images below. The amount of increase can be customized with the following variables.

; =====Change=size=of=GUI=when="Make Bigger"=is=invoked=====

HeightSizeIncrease := 300 ; Numbers, not 'strings,' so no quotation marks.

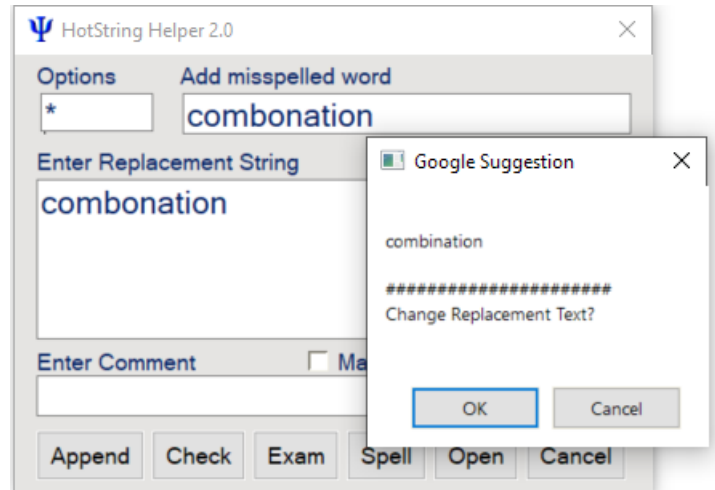
WidthSizeIncrease := 400



AutoCorrect word entries

As previously indicated, the above multi-line boilerplate template features are a big part of the hh2 tool. The *other* big part is the AutoCorrect multi-match creation and entry tools. Typically, the hh2 tool is activated by first selecting a bit of text, then pressing the hotkey (Win+H by default). The tool attempts to determine if you are creating a *boilerplate item*, -or- an *autocorrect item*. There is also a *check-for-existing-hotstring* mechanism, [discussed below](#). The check for template vs. autocorrect is as follows: If the selected text contains three or more space “ ” characters, or if it contains any new line “\n” characters, then assume it is a boilerplate template item and extract the initials from the first X words, of Y length. Also, if it is a boilerplate template item, use the settings defined in the variables, “DefaultBoilerPlateOpts, myPrefix, and mySuffix.” If there is text selected that doesn’t match this, then assume an autocorrect entry, and use the setting defined in the variable just below those, called, “DefaultAutoCorrectOpts.” If no text is selected, just open a blank hh2 form.

With the example screenshot on the right, I mis-typed “combination.” It occurred to me that this might be a good candidate for an autocorrect item, so I selected it with my mouse and activated hh2 via Win+H. To get the correct spelling for the replacement box, I then clicked the *Spell* button. The *Spell* tool is not a “spell checker” per se. It sends the word to Google Search, then returns Google’s “Did you mean....” response. Google doesn’t always provide a suggestion, but it usually does. The suggestion appears in a message box, as shown in the image. Clicking OK will update the text that is in the Replacement String edit box.



Note: My old eyes have trouble seeing the text in the form,

so I added a **Large Font toggle**. In the image the font is toggled to be larger. Please do not confuse the function of the “Make Bigger” button, with changing font size. The font is changed in one of two ways: (1) Ctrl+Up Arrow/Ctrl+Down Arrow or (2) Ctrl+Mouse Wheel Up/Down. The default smaller size is s11, and the larger size is s15. This is not a gradual “zoom,” it’s a toggle.

Auto Entry of New AC Items

With the above example, I had been typing a document (this document, actually) and double-clicked the misspelled word, and pressed Win+H to activate hh2. Then I corrected the spelling of the replacement string and added the item with the Append button. In a situation like this, once the hh2 form is closed, the original typo will still be selected. As such, we can automatically type the correction right into the document. The following variable assignment appears near the top of the code:

AutoEnterNewEntry := 1 ; 1 = yes, add. 0 = no, I'll manually type it.

If/when the value is 1 (one), hh2 will remember what the active document is, then when the document is active again, it will copy the selected text a second time, and compare it to the trigger of the just-created autocorrect entry. This ensures that the item in the document is still selected and the beginning/end of the hotstring item was not trimmed (I.e. to make a “multi match” item.) If the whole trigger was used, then it is assumed that the whole *replacement* was also used, so that is typed into the document via *SendInput*, replacing the selected text. Disable this feature by setting the variable to 0 (zero).

Word Analysis

The word “Analyze” is too big to fit on the button. I thought about having it say “Anal,” but ultimately, I decided on “Exam.” Click the Exam button to examine the word. When you do, the bottom part of the form, the *Exam Pane*, “pops down” and the Exam button text changes to “Done” as seen on the right.

As suggested in the [above section](#), “*Philosophy of best multi-word matches*,” the main goal here is to simultaneously “winnow down” the trigger and the replacement words, while also testing them as word -beginnings, -endings, and -middles, to figure out what combination gives us the most “bang for our buck” in terms of being a high-utility multi-match autocorrect entry.

When the *Exam Pane* of the hh2 form is expanded, the word-beginning/ending/middle options from the Options box will be propagated down to the Exam pane radio buttons. Changing the radio buttons will in turn update the Options box. Tip: Right-clicking any of the radio buttons will set them all to false/unselected, and the * and ? will both be removed from the Options box.

With my default setting of asterisk, you can see that there are already three potential words that would be corrected with the hotstring

```
.*:combonation::combination
```

The next step will be to progressively trim-off letters from the beginning and/or ending, using the [>>] and [<<] buttons. Clicking the wide Undo button (or pressing *Ctrl+Z*) will undo the trims until you are back to the original word. *Shift+Click* (*Shift+Ctrl+Z*) will undo all the trims, i.e. Reset the word.

Delta String

Please note the *blue delta string*: **comb [o | | i] nation**. This serves the purpose of giving us a quick visual of how many characters can be trimmed from the beginning and end before the “typo part” of the trigger is removed. The delta string has four components:

Characters that are -common to the beginning of both strings [-specific to trigger | | -specific to replacement] -common to the ending of both. Also look at this as **[the error | | the fix]** and at the ends are the parts that are neither error, nor fix.

The bottom-most part of the form has the two lists of word matches. The words that correspond to the trigger string are on the left, and the ones for the replacement string are on the right. The goal is to have *many* words represented on the right, and *none* on the left. The example in the screenshot will potentially fix three different words and won’t misspell any. If the “Misspells” box contains any words, then the Trigger String label will turn **red**. And will show how many words are matched. This allows the user to be alerted to misspellings, even when the Exam Pane is closed.

It is noteworthy that the Exam Pane is not useful for creating boilerplate template entries. Also, the Make Bigger feature is not useful for autocorrect entries. As such, there is no reason to have both active at the same time. Indeed, *making the hh2 form bigger will hide the Exam Pane, and vice versa*.

The screenshot shows the 'HotString Helper 2.0' application window. At the top, it says 'Options' and 'No Misspellings found.' Below this, there's a text input field containing 'combonation'. To the right of this field are buttons 'Make Bigger' and '+ Symbols'. Below the input field is a larger text area also containing 'combonation'. Below that is a section for 'Enter Comment' with a checkbox for 'Make Function'. At the bottom of this section are buttons: 'Append', 'Check', 'Done' (highlighted with a blue border), 'Spell', 'Open', and 'Cancel'. Below these buttons is a display area showing 'comb [o | | i] nation' with navigation buttons '>>' and '<<'. Below this are radio buttons for 'Beginnings' (selected), 'Middles', and 'Endings'. Below the radio buttons is a button labeled 'Undo (+Reset)'. At the bottom, there are two lists: 'Misspells [0]' and 'Fixes [3]'. The 'Fixes' list contains three items: 'combination', 'combinational', and 'combinations'. At the very bottom of the window, it says 'GitHubComboList249k.txt'.

Trimming

With the screenshot on the right, the top part shows the effect of trimming five letters off of the right side and having the *Middles* radio button selected. There are 40 potential word fixes and 0 misspellings.

The bottom part of the image shows the effect of then trimming one character from the left. The resulting trigger string ("ombon") is present in the word "trombone" and several other related words. So it was trimmed too much. I clicked the Undo button, then Appended it to my hotstring library file with the Append button. Cool.

Add-a-Letter

Sometimes you need to add a letter back to the beginning or end. An example happened when I tried to type "Lemmings" and it got erroneously changed to "Lemings." To remedy, I selected the word, pressed Win+H, and ran a Validity Check to get the line number of the bad entry. It was this item:

`.*?:emmin::emin`

I like to use the word "Lemming" sometimes, so I had to fix this. Upon expanding the Exam Pane, I saw that emin fixes 299 words, but emmin is needed for "lemming," "stemming," and a few others. We can't have that. So, I analyzed the words that contain "emin" and found that the letter "f" precedes "emin" in 80 of them (such as "feminine"). So I got rid of ".*?:emmin::emin" and added ".*?:femmin::femin" as well as a couple of others that would correct multiple words, but not misspell any. To automate the **Add-a-Letter** process a little, hh2 will watch the trigger string box, and if a letter is added to the beginning or end of the string, the same letter will be added to the replacement. An important point is that this behavior won't be desired when creating Boilerplate Template entries. As such, the Add-a-Letter feature is only available when the Exam Pane is open. The process of updating the replacement is slow, so add the letters slowly, or it won't keep up.

Important points:

- They must be added to the very left or right.
- Add the letters slowly.
- Only use this feature when the Exam Pane is showing.

Tip: There is a shortcut key... Pressing **Shift+Left** focuses the Trigger string box and puts the cursor in the Home position.

Known Problem: The add-a-letter effect is unreliable. Sometimes it just doesn't work. I haven't been able to figure out why.

Check For, and Capture, Existing HotString

The above "combonation" example involved selecting the typo with the mouse, then pressing Win+H to copy the word into the hh2 form. It takes time for you to Examine/Analyze a word though. If you are in a hurry, just use the Spell function to correct the spelling, then save the item as a whole-word autocorrect entry. *Then later...* When you have



time, launch hh2 and press Open. This takes you to the bottom of your autocorrect list, where the new items are. *Select the entire hotstring (including colons) and press the hotkey.* There is a “check for existing hotstring” mechanism. If a hotstring is found in the selected text, the trigger, replacement, options, and comment will be captured, and get populated back into the hh2 form for analysis. This is done by parsing the existing hotstring with a *Regular Expression*.

This particular [regex was created](#) by Andymbody and is optimized for efficiency, utility, and ease of use.

The Comparison Word List

As mentioned above, I’m a big fan of the WordWeb Dictionary PC application. While creating this manual, it occurred to me that I should check with the WordWeb developer about the legality of me sharing hh2 with a word list that was generated from his (commercial) software. Indeed, I learned that the list is “non reproducible,” so I’ve obtained several other similar, *open source*, word lists that will be included in place of the list that was used with the above screenshots.

There is a small text label at the bottom of the Exam Pane, that indicates what the currently assigned word list is. ([See screenshot](#) on a previous page.)

The to-be-used list of comparison words gets assigned via the “WordListFile” variable, near the top of the code, where the other user options are. Please include the text file extension in the name—such as: **WordListFile := 'GitHubComboList249k.txt'**

Note that the word list is also used by the [ManualCorrectionLogger](#) script. It is technically okay for the mcl and hh2 to used different word lists, but I recommend using the same list for both. This will allow the MCLogger validation feature to work better with your library of autocorrect hotstrings.

The Row of Six Buttons

Of the six buttons at the bottom of the main form, several have been discussed above. Additional information is as follows:

Append: It constructs a new hotstring from the information in the form and appends it (pending validation) to the user hotstring library script, then reloads the script, so that the new hotstring is ready to use. Note that pressing **Enter** when the cursor is in the trigger or options boxes will do the same. Notes: **Ctrl+Click** on the button will append the item, but not close the hh2 form, and not reload the script. Use this if you plan to add several hotstrings in succession. **Shift+Click** on the button will—rather than appending—construct the hotstring and save it to the Windows Clipboard. Remember that, when sending the string to the clipboard, if the Validation Message Box appears, you’ll need to press Shift again, when clicking “Append Anyway.”

Check: Do a Validity Check, but don’t append the item. The Check button does a *validity check* of the hotstring (whether boilerplate item, or autocorrect item), and reports information about it. Validity Checks are discussed in the next section.

Exam/Done: Opens the Exam Pane for [word analysis](#). Bonus tip: **Right-Click** the Exam button for the [Secret Control Panel](#). :- }

Spell: Uses Google’s “Did you mean...” to attempt a spell check of the text in the replacement string box.

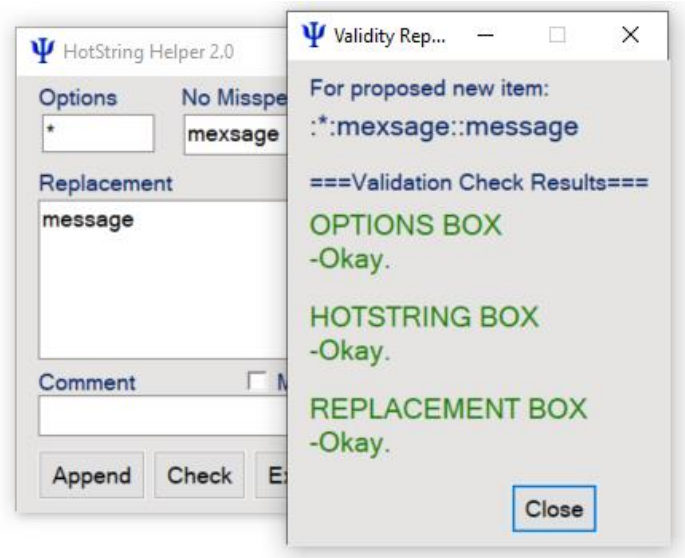
Open: This is for reviewing your recently-appended hotstrings. The Open button will open your AutoCorrect.ahk file in your default ahk editor and browse to the bottom of the file, where any newly-added hotstrings are likely to be.

Cancel: The Cancel button simply hides the hh2 form and restores the previous clipboard contents.

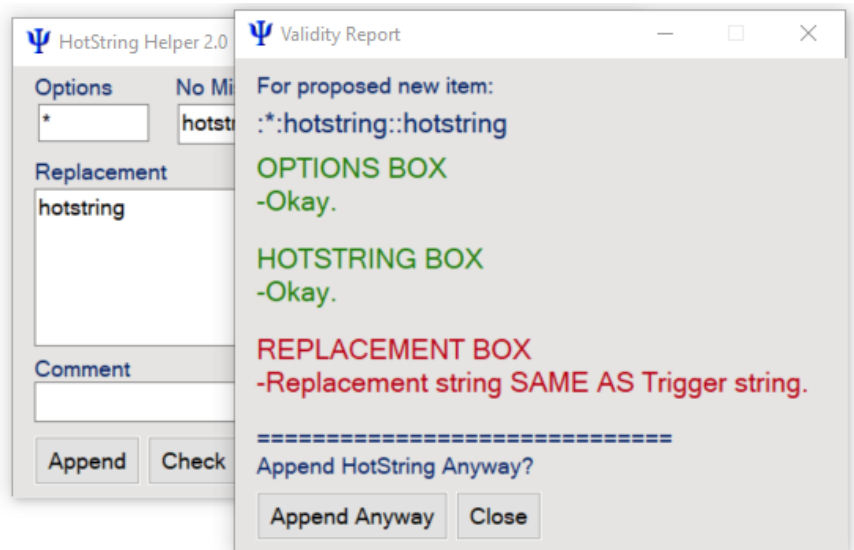
Validity Checks

The hh2 tool does several *Validity Checks*. Validity checks occur under two situations:

(1) If the user presses the Check button, a check is done and a message dialog such as the one on the right, will provide feedback as to whether there are concerns, or if the Options, HotString, and Replacement Boxes are all “Okay.” With the image on the right, they are all “Okay.”



(2) If the user uses the Append button, a check is also done, but a message is *only* displayed when there are validity concerns. There is an option to “Append anyway.”



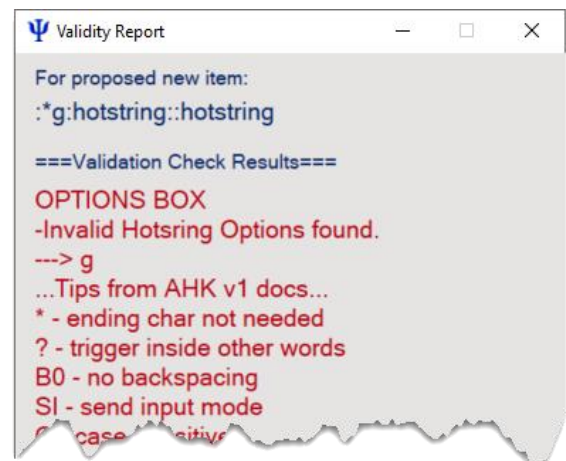
The validation checks, in order of occurrence, are as follows:

OPTIONS BOX.

Valid AutoHotkey hotstring options include such things as

* ? B0 X T C

and appear between the first double colons. The colons, themselves, are not options and should not be included in the Options box. The code will flag any characters that are not valid options (such as “g” in the screenshot to the right). It will also provide some brief *Tips* that are based on the [AutoHotkey Documentation](#).



HOTSTRING BOX.

Most of the validation mechanism is dedicated to checking the trigger string, i.e. the “hotstring.”

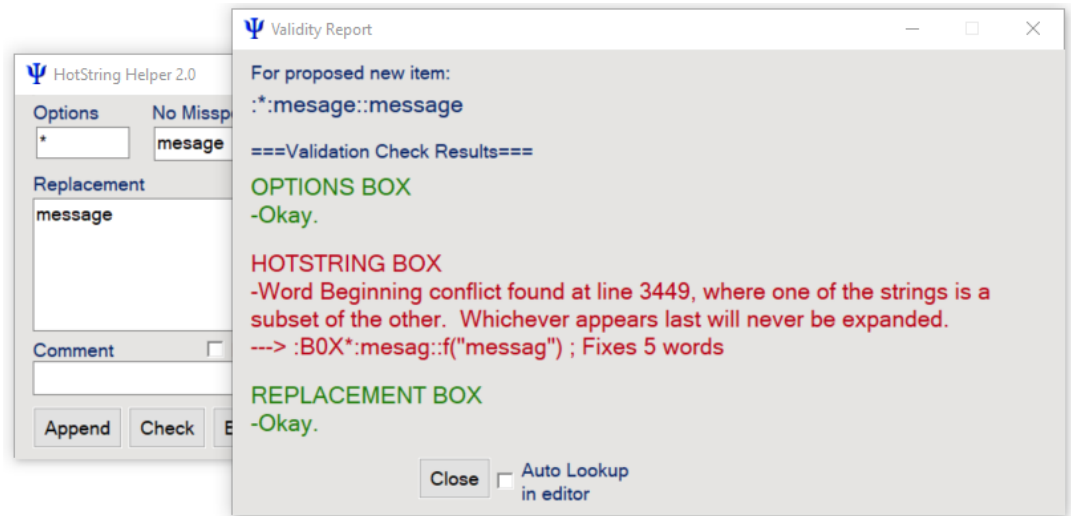
*If the trigger string edit box is blank, it will warn you.

*If you are about to create an entry with a trigger that is a *Duplicate* of an existing trigger, it will warn you.

*If the new trigger string is one of the four possible subset/superset conflict scenarios (such as the screenshot on the right), it will warn you. There is more discussion about these conflict scenarios below. Currently (April 2024) hh2 will identify the “*special situations*” for word endings discussed below, but it won’t advise what to do with the word-ending-match pairs.

*If the new trigger string is likely to inadvertently misspell other words it will warn you.

*If the new trigger matches a hotstring that was previously removed, it will warn you (but only if the string is in the ‘RemovedHotstrings’ list).



As seen in the screenshot, if (and only if) there is a problem with the hotstring which might require looking it up in your hotstring library, a checkbox will appear to “Auto Lookup in Editor.” To understand the functionality of this, please note, first, that the text displayed in the OPTIONS, HOTSTRING, and REPLACEMENT BOXes can be selected with your mouse. I made it this way, because, when processing potential multi-match autocorrect entries, I found that I often want to search my AutoCorrect2 code for the already-existing, potentially-conflicting string, or go to the relevant line number. In these situations, it’s much easier to copy the item to the clipboard, then paste it into the Find box of my ahk editor. In practice I find that I only ever copy text from the HOTSTRING BOX part of the validity dialog. To automate this “look up” process even further, I added the following rules: If you click on HOTSTRING BOX text (and therefore the focus changes to that part of the form), then a function *FindInScript()* is immediately called. That function waits for you to release the mouse button. When you do, whatever text is selected, gets saved to the clipboard. Then the tool attempts to open your assigned editor (assigned via the variable *MyAhkEditorPath*). The hotkeys are then sent to search for the text, or go to the line number. If the characters selected were digits, then Ctrl+G is sent, otherwise, Ctrl+F is sent. (Please note that these are the default hotkeys used by VSCode, other editors might use different hotkeys for “find” and “goto.”) So... The [] *Auto Lookup in editor* checkbox must be checked for this auto-lookup to occur. The checkbox is unchecked by default, but this can be changed, as described below.

REPLACEMENT BOX.

With the replacement box, the following will result in a warning:

*The string starts with a colon.

*The box is blank.

*The string is identical to the trigger string (as in a previous screenshot).

The user options for the Validity Message Dialog Box are near the top of the code, and are as follows:

```
;===Change=Settings=for=Big=Validity=Dialog=Message=Box=====
myGreen := 'c1D7C08' ; light green 'cB5FFA4' (for use with dark backgrounds.)
myRed := 'cB90012' ; light red 'cFFB2AD'
myBigFont := 's13'
```

AutoLookupFromValidityCheck := 0 ; Sets default for auto-lookup of selected text on
; mouse-up, when using the big message box.
; WARNING: findInScript() function uses VSCode shortcut keys ^f and ^g.

Conflict Scenarios

Our previous [philosophical discussion](#) of what makes a good hotstring might have also included, “*Is not a duplicate of, and does not conflict with, an existing hotstring.*” Consider the following pairs of (fake/sample) autocorrect entries.

```
; Word Beginning conflict (nullification?):  
; wizard2 never gets used, because typing "wizer" activates wizar1.  
; The order of these do not matter.  
:*:wizer::wizar1  
::wizerd::wizard2 ; Is nullified.  
  
; Word Middle conflict (nullification?):  
; Just like word Beginning conflict...  
; experiment1 never gets used, because typing "expire" activates xperi2.  
; The order of these do not matter.  
::expirement::experiment1 ; Is nullified.  
:*?:xpire::xperi2  
  
; Word Ending conflict:  
; Typing "likour " will result in one, or the other, WHICHEVER is first.  
; The second/bottom one is never used. Order matters here.  
::likour::liquor1  
:?:kour::quor2
```

First let’s point out that there might be legitimate reasons for duplicate replacement strings. For example, the two next items appear in the original 2007 AutoCorrect.ahk.

```
::teh::the  
::hte::the
```

These are both common ways to misspell “the.” They are not duplicate hotstrings, because the triggers are different. Now consider these two:

```
::theer::there  
::theer::three
```

These fix different words, so the autocorrect items, *in whole*, are not duplicates, but the duplicate *hotstrings* conflict with each other. Whichever appears first in the script file will be loaded into RAM by AutoHotkey (v1 or v2) and made available for use. The second one will be ignored. Regarding the fake items above: With the first pair, there is a word-beginning item (opts: *), whose trigger is a *substring* of the no-options one near it. Importantly, the substring is a “left-match” of the full string, i.e., the superset has additional characters *at the end* of the trigger string. These characters at the end will never be seen by AutoHotkey, because typing the first part of the trigger activates the “substring-match” item with the asterisk in the options (remember, * = no End Char is needed to activate the hotstring). The next pair has a word-middle (opts: *?), which works similarly. With either of these, the order doesn’t matter. The shorter one will always supersede the longer one *unless* there is a *Context Sensitive* section in your script. We’ll briefly look at Context Sensitive hotstrings [below](#), but to conclude this section, just remember that word-beginnings and middles don’t play nice with other hotstrings that are supersets. I’m not even sure that this should be called a “Conflict.” It’s really more of a “*Nullification.*” Word-endings on the other hand,

```
:?:toin::tion
```

actually do conflict with each other, in the sense that “*Whoever is first, wins.*” The order does matter. It is also relevant that, under certain circumstances when working with word-ending hotstrings, (1) conflicting items can both be used, and (2) potential misspellings can be nullified.

Special situations with word-ending multi-match autocorrect entries

The previous section introduced how word-ending-match autocorrect items can conflict with each other in the sense that the second one is never seen. There are exceptions. Consider the below pair of items from my current AutoCorrect list.

```
:?*:ngiht::night ; Fixes 103 words
:?*:iht::ith ; Fixes 560 words
```

In my own list they are embedded in f() functions ([discussed below](#)), and they are not right next to each other. They *are* in the same order though! The number of potential fixes reported, will depend on the word list used. Note also that as word-endings, there would be much fewer potential fixes. The 103 and 560 totals assume that these items are word-middles. The *relevant point here*, is that **when** there are “overlapping” word-ending items, where one is a subset, and is a right-most-match, **if** the longer one is first, then they both will be active. You can paste these into an .ahk file and experiment if you want (exit AutoCorrect.ahk first, if it’s running). Now (mis)type “goodngiht” and you’ll get “goodnight.” (mis)Type “wiht” and you get “with.” However, if the shorter substring item is first, then the longer one doesn’t work. In that case, (mis)typing “goodngiht” yields “goodngith”. The hh2 Validity check mechanism is not detailed enough to recognize these intricacies. It will just report, “Word-ending conflict.”

The other special situation works similarly. In this case, the longer word is one that we want to preserve and prevent it from getting misspelled. The items below are adapted from Jim B’s original 2007 AutoCorrect.ahk.

```
:B0:design:: ; The B0 option turns off backspacing,
:B0:feign:: ; so the typed word just stays there.
:B0:resign::
:B0:sign::
:B0:sovereign::
{ ; <--- braces needed for v2, but not v1 AHK.
    return ; This makes the above hotstrings do nothing
} ; so that they override the ign->ing rule below.
:?:ign::ing
```

Notice the word-ending fix for “ing” at the bottom. It’s interesting that there have been several articles and blog posts about the original AutoCorrect.ahk script over the years. It usually gets described as “*The script fixes some 4700 common misspellings and typos.*” In reality though... If you count the ign -> ing fix, the number of corrections is much higher! Depending on what word list you compare against, the number of potential corrections might be from 7 to 15 thousand just for that one hotstring.

Unfortunately, it also misspells 40 or so words, such as “sign.” (Not all are shown above.) With the above chunk of code, you’ll notice that, again, the longer-string-items appear first. The “B0” option tells AutoHotkey to not remove the trigger string. And since there’s no replacement, the typed word just remains unchanged. Typing “ign” by itself, or following any characters that are not one of the preceding longer strings will trigger the replacement. The longer, preceding strings are “protective” in the sense that they protect us from “miscorrecting” the words. A relevant topic that is related to this is the topic of “sacrificial words.”

When misspellings are tolerated

As discussed above, a good autocorrect entry corrects lots of common words, but *doesn’t misspell other words* in the process. But what happens when a potential entry corrects many common words, but misspells one or two **uncommon** words? Do we sacrifice that word; knowing that if we ever type it, we’ll have to press Ctrl+Z to undo the (mis)correction? Consider this example:

```
:?*:ahve::have ; Fixes 47 words, but misspells Ahvenanmaa, Jahvey, Wahvey, Yahve, Yahveh (All are different Hebrew names for God.)
```

The entry “::ahve::have” is one of the whole-word AutoCorrect 2007 items. By simply adding the “?” it matches 47 English words. As it turns out, there isn’t a single English word with “ahv” in it. Unfortunately, that three-letter combination does appear in several of the Hebrew names for God. If a person is a Jewish scholar (that writes in English), they will definitely be interested in this dilemma.

Here is another example:

```
:?:itr::it ; Fixes 366 words but misspells Savitr (Important Hindu god)
```

By using the same technique that Jim did with “?:ign::ing” we can “protect” some of these words. Unfortunately, it only works with word-endings. It does not work for word-middles or -beginnings (i.e. Ones that have an asterisk in the options.) So, this uncommon word:

```
:B0:Savitr::
```

gets protected. However the words,

```
:B0:Ahvenanmaa::  
:B0:Jahvey::  
:B0:Wahvey::  
:B0:Yahve::  
:B0:Yahveh::
```

don't get protected because the corresponding hotstring “?:*:ahve::have” doesn't allow it. I put all of these near the top of the autocorrect list, then commented-out the ones that won't work. There are 37 “good” protective strings and 35 commented-out “no good” ones. Fortunately, they really are pretty obscure words. But, as indicated in the code comments: ***“If you hope to ever type any of these words, locate the corresponding autocorrect item and delete it.”*** Both sub lists are sorted alphabetically. For the above examples, I took the first item from each. It was just a coincidence that both “sacrificial” words were religious words. But... Since we're doing religious words...

Using Case Sensitivity to protect words

Consider this example:

```
:*C:carmel::caramel ; Fixes 12 words. Case sensitive to not misspell Carmelite (Roman Catholic friar)
```

This was made from the 2007 whole-word entry, “::carmelite::Carmelite”. Believe it or not, there are 12 English words that have caramel as a root at the beginning. Adding the asterisk allows the hotstring to match any of them. Unfortunately, a Roman Catholic friar is (apparently) called a “Carmelite.” I looked it up in the dictionary, and it appears to always be capitalized. So, we add the “C” option. Now if we type the word with a lower-case “c,” which matches the trigger, it gets changed. If we use upper case, it does not get changed. Unfortunately, this also means that **if** I write a sentence about caramel, **and** I start the sentence with the word caramel, **and** I misspell it as the trigger string, it won't get corrected. I guess that's another “sacrifice.”

For most purposes, this *case sensitivity to protect words* technique only works with word beginnings. When the to-be-protected word is an acronym (or initialism), then word-endings or -middles might also apply. Here is an example:

```
:?C:hc::ch ; Fixes 446 words, :C: so not to break THC or LHC
```

This one is “safer” in the sense that a misspelled word at the beginning of a sentence will still get corrected; Unless, of course, we are writing in all caps. An opposite example might be:

```
:C:ASS::ADD ; Case-sensitive to fix acronym, but not word.
```

As a school psychologist who often writes reports for kids with Attention Deficit Disorder, this autocorrect can prevent a very embarrassing typo. On the other hand, if I were an inflamed internet troll, I may want to remove this item from my list, so that I could blast people, by shout-typing obscenities.

Context Sensitive Hotstrings

Two pages back, when discussing the problem of duplicate hotstrings, we mentioned Context Sensitivity. AutoHotkey does allow us to create [Context Sensitive](#) hotstrings and hotkeys. A self-explanatory example right out of the documentation is as follows:

```
#HotIf WinActive("ahk_class Notepad")  
::btw::This replacement text will appear only in Notepad.  
#HotIf  
::btw::This replacement text appears in windows other than Notepad.
```

The hh2 app does not create Context Sensitive hotstrings. Also, it is important to note that the above-discussed validity checking will not recognize if a hotstring is embedded between #HotIf directives. It will report, “Duplicate found,” even though duplicates are

okay, when one is restricted to certain contexts (which are usually particular windows). There is, however, a workaround... Just before the hotstring definitions, there is a line of code with:

```
getStartLineNumber := A_LineNumber + 10
```

Validity checks only happen below this point, so have the #HotIf items above it.

Inevitable Errors

Despite our best efforts to create hotstrings that don't have triggers corresponding to other words, there will inevitably be times when errors will occur. From my own experience, I believe there are two scenarios in which this happens the most. Keyboard input buffering problems and confounding mis-typings.

Regarding keyboard input buffering

Consider this test hotstring:

```

:~*:zX::|||||

```

Put this in a script and try it. You'll notice that "zxcv" are next to each other on the keyboard, so you can type them rapidly by "strumming" the fingers of your left hand. Try it several times... This is what I get:

The AutoHotkey application is written such that hotstring execution is meant to be buffered. This means that, once the hotstring is activated, any additional keypresses should be held in RAM, then sent *after* the hotstring is finished with its own characters. The resulting expanded text *should* look like this:

Figure 1 shows four schematic representations of data sets, labeled CV, C, S, and T. Each representation consists of a horizontal bar divided into segments, with a vertical line indicating a specific point of interest. The segments are labeled with numbers 1 through 10, and the vertical line is labeled with a letter (C, S, or T) corresponding to the data set.

Communication with the current AHK developer, *Lexikos*, suggests that other applications can compete for the keyboard buffer and cause the effect of interspersed letters. This example has a really long replacement string, but it can happen with shorter strings as well.

I find that, when expanding a boilerplate entry, I tend to sit there and wait for it to play back before I continue typing. *With autocorrects, however*, a person doesn't know when they will occur, so there isn't really an ability to pause for it... You just keep right on typing, which invites buffering glitches. As discussed in the [Log Analysis](#) section, below, a special tool has been created by forum member, Descolada, which very well addresses this input buffering dilemma.

Regarding confounding mis-typings

Consider this autocorrect word-middle item:

:?*:cirp::crip ; Fixes 126 words, but misspells Scirpus (Rhizomatous perennial grasslike herbs)

Now imagine I intend to type “chirp,” but I mistype it, “cirq.” Since that error string happens to be a trigger for a different word, my AutoCorrect tool erroneously changes it to crip. “*I heard the robin crip.*” I think this word-middle autocorrect entry was probably generated from the 2007 whole word item, “::script::script.” There’s a trade-off here, shorter trigger strings (as word parts) match more words. But... longer trigger strings are less likely to be accidentally typed during a confounding misspelling. Which one should be used?

-longer strings = more unambiguous

-shorter strings = more potential matches

An argument can be made for each, but for the most part, I leaned more toward shorter strings with more matches, when processing the 2007 AutoCorrect and the Wikipedia grammar items. It's also noteworthy that shorter strings will correct the typo sooner. This is important because, if I notice the typo, I'll manually correct it. So it's nice if the autocorrect script fixes it first. Also though, it bothers me when my AutoCorrect script makes changes and I can't keep track of, or even detect, them. I needed a way to capture the corrections.

AutoCorrection Logging

I don't know who the first person was to use *Function Calls* for individual AutoCorrect items, but I [got the idea](#) from Mikeyww. The dilemma was how to capture the most recently used trigger into a variable so that it could be viewed. The AutoHotkey application has several built-in variables (**A_Vars**), that capture various metadata about running scripts. In fact, there is an **A_ThisHotkey** variable that contains the most-recently-used hotkey. It will also work for hotstrings, but... not for *auto-replace* hotstrings, which is exactly what is needed for our purpose. So how do we collect that information? The solution is a function-calling hotstring. It won't work if you have this:

```
::tpyo::f("typo")
```

because this will only replace "tpyo" with the text "f("typo")." However, adding an "X" in the options tells AutoHotkey to treat the "replacement" section as computer code. It's "X for eXecute."

This version:

```
:X:tpyo::f("typo")
```

Will indeed call the function (assuming there is a function f() defined somewhere). Remember that we want to capture that last used hotstring trigger. By adding the X option, this is no longer an *autoreplace* hotstring, so we can send the variable content to the function to be used.

```
:X:tpyo::f("typo", A_ThisHotkey)
:X:hello wrold::f("hello world", A_ThisHotkey)
f(replacement, trigger)
{
    Send replacement
    Last_used_trigger := trigger
}
+!F3:: MsgBox Last_used_trigger
```

Now press **Shift+Alt+F3** to see the last used hotstring trigger. Nice. It occurred to me later that I needed a third parameter, to hold the *End Character*, so that it could be Sent, following the replacement text. Rather than simply hold (only) the last-used trigger string, it also occurred to me to append them to a Log File for later analysis.

The AutoCorrectsLog File

When an auto replacement is made, the function beeps, then waits for one second, and logs the item by appending it to the bottom of the *AutoCorrectsLog.txt* file. One line per item, formatted with the date as YYYY-MM-DD. The reason for waiting one second is to see if the user presses the Backspace key. This information is recorded in the log file. If there is a normal double-hyphen "--" separating the date and string, then Backspace was not pressed during the one-second timeout. If there is a left arrow "<<" then Backspace was pressed.

Example:

```
2024-01-19 << :?*:voiu::viou
2024-01-19 -- :?*:cuas::caus
```

The assumption here, is that, if an autocorrect item "goes rogue" then you'll immediately press Backspace to correct it. Unfortunately, this is not a perfect system. For example, when I'm tired and busy and distracted, my typing can get pretty sloppy. It is entirely possible that the autocorrect will make the desired correction, but I'll also make an *additional* error—and backspace it—

all within one second. Also, it's noteworthy that I tend to watch the computer screen as I type. This is true for most people, but not all. I suspect that most people who, for whatever reason, don't monitor their typing, probably won't want to use an autocorrect app.

It is noteworthy that the `AutoCorrectsLog.txt` file is not written-to after every autocorrection. Instead, the to-be-logged items are saved in a cache, then written in-bulk on an interval that is set near the `f()` code. It is also noteworthy that, if the cache is empty, file appends will temporarily stop. They will automatically start again when the cache again contains items to be logged. The variables for this are as follows:

```
saveIntervalMinutes := 20
IntervalsBeforeStopping := 2
```

After logging quite a few autocorrections, I wanted to generate frequency reports to analyze the data. At first I had embedded the code for that right in the log file. In Oct 2024 the log was put in a separate text file called `AutoCorrectsLog.txt`. I don't recommend renaming any of the supporting files in the `AutoCorrect2` package.

AC Log Analysis

It is noteworthy that the `AcLogAnalyzer` script does not run in the background like `AutoCorrect2` and [ManualCorrectionLogger](#) do. The analyzer must be run manually, though there is a shortcut in the hh2 Secret Control Panel (right-click Exam button).

The image to the right shows what the reported information looks like. The items are sorted by how many times they were Backspaced. Only the top X items are reported. The number of times the same item was *not* Backspaced is provided for comparison. The item,

```
Found 4<< and 9--for :*:dont::don't
```

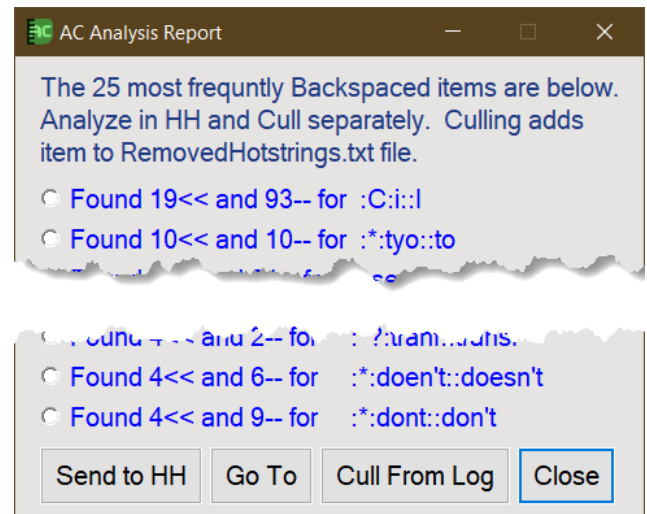
means that this autocorrection was logged 13 times. Four of those times, a Backspace occurred within one second. The other nine times, it did not.

If a hotstring is followed by a backspace many more times that it is not, then it is probably a good candidate for removal. It is noteworthy that items can sometimes be *changed* rather than being fully removed. For example, I used to have the word-middle item `:*?:daty::day`. The analyzer indicated that it had been removed seven times, and only kept once. Presumably, it was getting triggered when it was not supposed to. That was unfortunate, given that it potentially fixed 168 different words. Rather than just deleting the item, I changed it into two separate autocorrect entries, a word-beginning and -ending:

```
:BOX?:daty::f("day") ; Fixes 48 words
:BOX*:daty::f("day") ; Fixes 72 words
```

That was a few months ago. We'll see if either item gets picked up by the analyzer in the future. PLEASE NOTE that the analyzer tool assumes you are using a version of `AutoCorrect2` that supports importing command-line arguments to HotString Helper. `AutoCorrect2` was set up to allow this feature on 5-4-2024. Please select an item via its radio button, and use the **Send to HH** button to have the item imported to HotString Helper2. Once the item has been fixed and/or removed from your hotstring library, you'll probably want to cull it from the autocorrection log (so you don't keep analyzing it). Use the **Cull From Log** button for that. ALSO NOTE that Sending to HH and Culling from Log are separate processes—So be sure not to cull it unless you remember to fix/remove the item from your library!

As with previous versions, *I'll leave my logged items in the log file for users to experiment, but obviously, a person will want to delete them and collect their own data.* It is your own typing (and mistyping) patterns that you want to analyze.

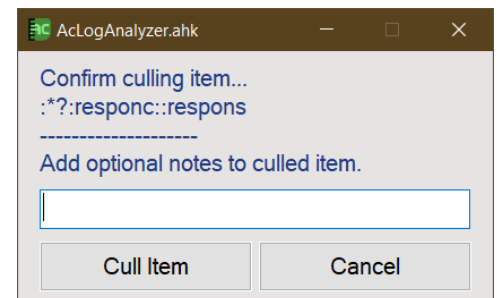


Running the analysis does take time... On my laptop, 1300 items takes about one-and-a-half seconds. I did a test run with 10k items though, and that took 44 seconds. Note the **SortByBS := 1** near the top of the code. If you change the value to 0 (zero), then the list will be sorted by “OK” (not backspaced) items.

The logging process is useful, but I have to admit that it is not the thing which has had the biggest impact on increasing the reliability of my system. I previously had two different logs, collecting different levels of contextual information. Even with all that data collected I was getting nowhere. I was still seeing enough errors that I had discontinued using a previous version of this script. Then, *Descolada* made his [InputBuffer Class](#) and shared it. This made a substantial difference, and I saw, perhaps, 80 or 90 percent fewer errors! After that, I readopted the current script and have been happy with it. The InputBuffer Class is now embedded in AutoCorrect for v2, and it gets used by the f() function. I did remove the *Mouse Buffering* elements from it, since the f() function doesn’t do anything with the mouse. The data logging and analysis is still important, but the buffering effect is way more important.

HotString Logging Circular Problem

At the time of this update to the manual (Nov 2024) I’ve been logging and analyzing AutoCorrections for about a year and Manual Corrections for about seven months. As indicated below, the [Manual Correction logger](#) looks for words that I’ve manually corrected. The input hook code was logging a lot of un-useful typing data. I lessened this effect by having it check to ensure that the captured correction doesn’t already exist in my AutoCorrect library. The point of logging manual corrections is to see if reoccurring typo-correction pairs should be added to my library—but there’s no point adding an item that is already there. As a side note: Theoretically, if a typo has a corresponding autocorrect item, then it should never need to be manually corrected, but it does happen when the manual correction is done (by me) before enough of the characters have been typed to trigger the autocorrect hotstring. Anyway... The MCLogger tool ensures that a typo-correction pair doesn’t already exist in the HotstringLib library before logging it. There was a hole in this system though. If the ACLogAnalyzer tool flagged an autocorrect item as bad, and I removed it, the MCLogger tool might then re-capture the same typo-correction pair, leading me to inadvertently reintroduce it to my library. To prevent this, I needed a way to permanently save a list of previously-removed items, so that I wouldn’t keep adding (and then needing to remove) them. In Oct 2024 I added the **RemovedHotstring.txt** file. This file holds items that are culled from the ACLog by the ACLogAnalyzer tool. The user can optionally add an in-line note to the removed item. This can be done using the “Cull confirmation” dialog that is seen on the right.



The RemovedHotstrings file is read by the MCLogger tool, to ensure that previously-removed autocorrect items are not offered-up as possible new items. It is also read by hh2’s [Validation Tool](#). If the user attempts to add a new hotstring via hh2 and the hotstring matches one in the removed hotstrings list, a warning is provided.

HotString Rarefication

While working with some of the longer “Grammar corrections” garnered from [Wikipedia’s list](#), it occurred to me that I could “**Rarify**” some of them to make them leaner... More efficient. Consider this hotstring:

```
::agreement in principal::agreement in principle
```

When auto-replacing a hotstring, the normal operation is for AutoHotkey to “Backspace away” the trigger string, then type the replacement. So, the above hotstring, as it is written, will instruct the AutoHotkey app to send Backspace 22 times, then type out the replacement. This is largely superfluous, because only the last two characters are actually different between the trigger and the replacement. So why not just press Backspace twice, then type the last two characters of the replacement string? We could turn off automatic backspacing with the “B0” (b zero) hotstring option and setup the autocorrect item like this:

```
:B0:agreement in principal::{BS 2}le
```

However, without the asterisk, an End Char is needed to trigger the string, so we need to “remember” what End Char was used and put it back. For that, we need to use the X option. Either of these will work:

```
:B0*:agreement in principal::{BS 2}le
```

```
:BOX:agreement in principal::SendInput "{BS 2}le" A_EndChar
```

The f() function, as it currently exists, will automatically, at runtime, compare the trigger and replacement strings and will only remove the necessary characters. The hotstrings don't have to be set up as in the above examples. This just happens automatically every time the function is called. It's noteworthy that in about December 2023, a feature was added, by Lexiko, to AutoHotkey v2, to streamline auto-replace hotstrings in a similar way.

The f() Function

The function as it currently exists, takes one parameter.

```
:BOX:trigger::f("replacement")
```

Previous versions used three parameters (as below), but upon studying [Descolada's code](#), I saw that the second two parameters are not needed, because they are essentially like "global environmental variables" anyway, and so are available to the function without the need to send them along with the function call.

```
:BOX:trigger::f("replacement", A_ThisHotkey, A_EndChar)
```

What the function does:

- Call the InputBuffer, which captures any manual keypresses.
- Save the hotstring to a variable so we can peek at it and/or log it.
- Rarify the replacement and determine the number of backspaces needed.
- Type the necessary backspaces and needed part of the replacement string.
- End the Buffer, thus typing any captured manual keypresses.
- Announce the replacement with a beep.
- Log the item, (via calling the GoLogger function) indicating whether Backspace was manually pressed by the user within one second.

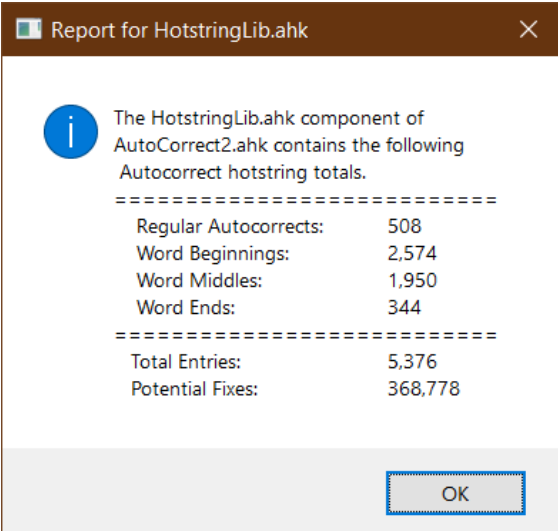
As seen in the above screenshots, there is a ☐ **Make Function** checkbox under the Replacement String box, but above the Comment box. Check the box if you'd like hh2 to format the new AutoCorrect entry with the f() components. If the box is unchecked, a normal hotstring will be created. Should the checkbox be checked by default? In the part of the code where the hh GUI is made, is the following line of code:

```
ChkFunc.Value := 1
```

That's a 'number one.' Change it to a 'zero,' or delete that line of code to cause the *Make Function* checkbox to be *unchecked* at startup.

AutoCorrect Fixes Report

In the code, before the list of AutoCorrect items, but after the InputBuffer Class, is a small utility to add-up a few things. Pressing **Ctrl+F3** provides the mini-report on the right. "Regular Autocorrects" refers to whole-word items. Beginning, Middle, and End items are the titular multi-match items. When I started manually converting the 2007 AutoCorrect list, to see how many multi-word matches could be made, I flagged some of the higher-utility items with "Fixes X words" as an inline comment. Upon creating the previously-mentioned Word Analysis GUI Tool, I automated the process of adding the flag. Once practically every item was appended with its corresponding number of potential fixes, it only made sense to automate the process of tallying those. Mikeyww helped with the original scriptlet. As of this update (Nov 2024) AutoCorrect2 can potentially fix >368k items, by using 5,376 autocorrect items.



Report for HotstringLib.ahk

The HotstringLib.ahk component of AutoCorrect2.ahk contains the following Autocorrect hotstring totals.

Regular Autocorrects:	508
Word Beginnings:	2,574
Word Middles:	1,950
Word Ends:	344
Total Entries:	5,376
Potential Fixes:	368,778

OK

Some Caveats

* The “368k” claim doesn’t mean that 368k unique words can be fixed. Indeed, the individual items were converted using the word list from the WordWeb app. The list had about 186k words... So how can there be 368k possible fixes? It is because there are several different *and common* ways to misspell any given word. Also, one word-middle item might fix part of a word, and another might fix a different part of the same word. So, there is a great deal of overlap between the words, in terms of which words can potentially be corrected.

* Another important point is that the *actual* total number of potential matches—and—the number of possible misspellings, will vary *greatly* depending on which word list is used as a comparison. Obviously a word list with 466k words will indicate more matches than the one with 26k words. It’s unfortunate that I used the WordWeb app when converting the bulk of the list, but then was not able to include that with the ac2 download. This means that, if you choose a multi match item from the library, capture it via Win+H, then check the Exam Pane, you are likely to find a higher number of matches than what is currently indicated in the items inline comment. Unfortunately, there would also be more potential misspellings indicated. What would the number of potential matches be if I had used the 249k word list to convert the entire list? $(249/186) * 368 \sim 492$, but I don’t know.

* It’s worth noting that I frequently reference Jim’s 2007 AutoCorrect.ahk list, which is based on the Wikipedia list, and the grammar items, based on another Wikipedia list. Indeed, perhaps 80% of the items came from those two sources. In reality though, the process of winnowing-down the words to create multi-match items loses the reference to the original word. So my version bears little resemblance to the original lists, and it would be impossible to reverse the conversion process, to get the original items back. As you might guess, *many* of the items in AutoCorrect 2007 were also removed, because there were redundant root words with different prefixes and suffixes. It was usually the root word (or part of it) that got used for the multi-match items, so the redundant whole-word items got culled.

* It’s also noteworthy that most of the grammar-fix items are comprised of multiple words. Since none of my word lists (which I used to calculate potential word matches/fixes) include multi-word phrases, there’s really no easy way to programmatically determine the number of potential fixes per item. As such, most of these items are just flagged with “Fixes 1 word.”

* It is relevant that the original AutoCorrect 2007 has 262 commented-out “ambiguous” items, such as, “::wich::which, witch.” What should the replacement be? “which” or “witch?” I chose “which” because it is a word I use more often. I also picked my preferred replacement for most of the other ambiguous items as well. Many of the ambiguous entries were ambiguous because of British vs American English. My apologies go to British users who keep getting “Americanized” autocorrections.

* The list is not perfect! In the “Rarefication” section, I use the example of

`::agreement in principal::agreement in principle`

Upon proofreading this manual, it occurred to me that

`::in principal::in principle`

would be a higher-utility item because it matches “agree in principal,” and many other phrases. I searched the list, however, and found that both items, in addition to

`::agree in principal::agree in principle`

were already present. The “in principle” one makes the other two superfluous. There are probably other superfluous or redundant items that I haven’t found yet.

* It’s hard to search your AutoCorrect.ahk file for the typos! If I open my script, then press Ctrl+F to “find” something, then attempt to type one of the trigger strings (assuming my script is running), it will be corrected. To avoid this, type half of the trigger, then press Left Arrow, then Right Arrow, then type the rest of the trigger string. This “breaks” the sequence. Other things will do this too, such as mouse clicks or pressing Esc. You can also just copy the trigger, then paste it into the Find box with Ctrl+v.

In The Script.

Some other things present in the script are as follows:

The CAsE CoRrector tool.

The original 2007 AutoCorrect.ahk script included a tool to *AUto-CORrect TWo COnsecutive CApitals* that was originally created by [Laszlo](#). It was a clever idea, but had problems, and so was commented-out by default in AutoCorrect 2007. As presented in the AutoCorrect for v2 thread:

I can't be the only one who has erroneously typed "THanks" more times than I've typed the intended "Thanks." It's a useful tool! The original was made by Laszlo, using a loop to monitor for hotkeys. It was a smart idea, but would misfire under certain conditions, and so is commented-out, by default, in the original AutoCorrect.ahk script. A version posted and discussed [here](#) was made by myself (with much help) using the InputHook() function. This was much better (I've been using it for months) but it introduced a couple of other limitations (as seen in the forum thread). Just a couple of days ago, forum member Ntepa posted an even better version (same forum thread). His uses a hotstring loop combined with InputHook and addresses the limitations of my own version, as well as adding a couple of additional improvements. It was so superior to mine that I removed my own version and included his (with permission) in the above code. Tip: Ntepa has cleverly included a 400 ms timeout because the double-capital effect usually occurs from typing too fast. If a person wants to experiment with the tool, they might like to temporarily remove the timeout. Do this by removing the T.4 from InputHook("V l101 L1 T.4"). You could also shorten/lengthen the timeout to fit your typing/typo speed.

Unfortunately, none of the above versions are completely error free. Even with the latest version, I occasionally have a mid-sentence proper noun erroneously converted to lowercase. I had logged the active window during erroneous conversions, but there didn't appear to be any particular applications that were correlated with the effect.

I previously had the CAsE CoRrector tool and AutoCorrect as separate script files for a while, but I had then re-combined them so that they could share the log function "GoLogger()." Since then, I've discontinued logging case corrections though.

Accented words with diacritic characters.

The AutoCorrect 2007 script includes 287 *accented words*, such as the example below. As with this example, many of the items have plural forms, so I made them word-beginning items. I don't consider these to be "typos or misspellings" so I didn't embed them in function-calls. I thought it would be fun to tag each item with its dictionary definition.

:*:angstrom::Ångström ; noun a metric unit of length equal to one ten billionth of a meter (or 0.0001 micron); used to specify wavelengths of electromagnetic radiation

Most of the definitions were obtained "in bulk" from <https://www.easydefine.com/>. Others are from the WordWeb application.

[Above](#), we discussed the "Show Symbols" button of HotString Helper 2.0. The *diacritic characters* in the accented words are similar to this, in that they require your .ahk file to be saved in "Unicode" format for the characters to be displayed correctly. As indicated in the [AutoCorrect for v2 forum thread](#), I recommend this:

- 1) Open plain old Notepad.
- 2) Do "Save As"
- 3) Change "Save as type" from txt to "All files (.)"
- 4) Change the Encoding to "UTF-8 with BOM."
- 5) Save (using .ahk extension) /close/reopen.
- 6) Paste in the code that contains the accented words and save again.

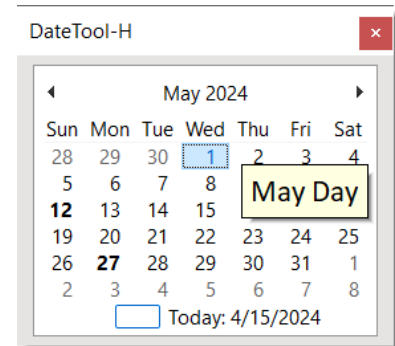
Other Bundled Tools

There are several other scripts that are included with AutoCorrect for v2.

Date Tool – H

~ The “H” is for Holiday :- }

The script called “*DateTool.ahk*” is a standalone tool in its own right, but it is incorporated with the AutoCorrect code byway of an **#Include** directive near the top of the code. If a person didn’t want to use the date tool, they could remove that #Include line of code. The default hotkey, **Shift+Alt+D**, shows the form as seen on the right. Yearly holidays are programmed into the code and appear as bold dates in the month calendar. Clicking on a holiday shows the name (again, see image). Pressing a number key (1-5) shows that many months. The default number of months can be set with the variable in this line of code:



```
MonthCount := 2 ; default number of months to display vertically
```

Pressing the “h” key toggles a list of all the holidays currently shown. They appear in a popup list as seen on the right. Pressing “t” takes the calendar back to “today.”

Credit for the creation of this tool goes almost entirely to AHK Forum user **Just Me**. I saw [an old post](#) by *PhiLho*, about bolding certain days in the AHK MonthCal GUI; and [another post](#) by *Tidbit*, that featured a function to predefine yearly US Holidays, and return whether a given date was on one of them. I had added a “help” post on the v1 forums to see if anyone could combine them. [Just Me did](#). Then later, a couple of us asked if anyone could convert the tool to AHK v2 code. Again, [Just Me stepped in](#) and converted the *isHoliday()* function and the MonthCal code for us. Check the *isHoliday()* function to see how to set up holidays. You might want to remove my wedding anniversary – LOL. I tweaked the code to allow single-occurrence events, though the tweak is kludgy. Some time later, I saw the [ToolTipOptions Class](#) – Again, created by *Just Me* – and added that to the bottom of the code. ToolTipOptions is what allows the nice big popup tool tips seen in the images. The font, color, and other things are customizable. Cool. “Holidays” that are one-time (non-repeating) events can also be added. See the comments in the code for instructions.

Jul-04	Independence Day
Aug-25	Anniversary
Sep-02	Labor Day
Sep-22	Autumn Equinox
Oct-14	Indigenous Peoples Day
Oct-31	Halloween
Nov-11	Veterans Day
Nov-28	Thanksgiving Day

In addition to the pop-up GUI Calendar, there is a hotstring component to DateTool-H. The rule for entering dates is: **;dn**. (Semi-colon, letter D, and n, a digit from 0-9, to enter a date which is n-days in the future. Therefore:

;d0 enters the current date.
;d1 enters tomorrow’s date.
;d2 enters the day after tomorrow.
etc.

The rule for entering dates in the past, is to use “double-Ds.” So:

;dd1 enters yesterday’s date.
;dd2 enters the day before yesterday
;dd3 enters two days ago.
etc,

A popup tooltip appears near the entered date, telling you the *Day of the Week* for the date you’ve just entered. If the day was a holiday, the tooltip shows the holiday (see image). If the date was “this week” or “last week” this is shown on the tooltip (also in image). If the date falls on a weekend, that too is shown (not in image). The date formats for the date that gets entered, and for the dates in the list of holidays, can be individually defined.

4-8-2024
Solar Eclipse ---> Monday last week

Manual Correction Logger (MCLogger)

The description “*manual correction logger*” is meant to convey that it *automatically* logs your *manual corrections* as you type. The script was made to record my own reoccurring manual corrections, so if any particular reoccurring typos keep happening, I can potentially make them into new AutoCorrect library items.

The script is mostly independent of *AutoCorrect for v2* or *HotString Helper 2.0*, and can be used even if neither ac2 or hh2 are used, though if no AutoCorrect Hotstring Library file is found, the append function will simply create one. There is a preference option, in the code, to either append the selected hotstring or to, instead, open it directly in hh2. Of course, AutoCorrect2 does have to be installed for this feature to work. AutoCorrect2 was set up to allow this feature on 5-4-2024, so use a version at least that new. The preference option is the variable called

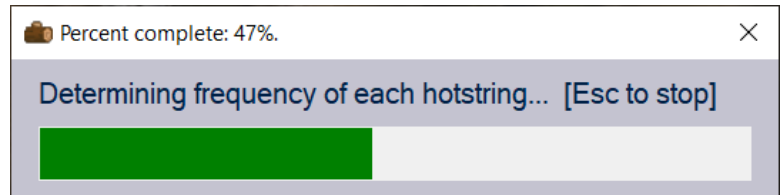
SendToHH := 1

Set it to 0 (zero) to disable the feature. Please note that the hotstring is sent to HH via a command-line parameter that is detected by AutoCorrect2.ahk. MCLogger has no way to confirm that the parameter has been received, so if you have “SendToHH := 1” but the receiving AutoCorrect file doesn’t have this feature, the hotstring will be lost. It will not be appended.

Note also, that MCLogger also uses the same WordList that is used by the Exam Pane in hh2 (though you can assign a different wordlist near the top of the code). There is also a simple validity check, against your library of autocorrect strings, “**HotstringLib.ahk**,” or whatever you call yours. And for previously-removed hotstrings “**RemovedHotstrings.txt**.” So the files needed for this tool are:


MCLogger.ahk
MCLogger.exe
MCLog.txt
GitHubComboList249k.txt (or whatever list you use)
HotstringLib.ahk
AutoCorrect2.ahk (if you want to use ‘SendToHH’)
RemovedHotstrings.txt

In addition to the progress bar seen in the above image, is the cull-and-append dialog seen to the right. The top X most-frequently-occurring strings are offered as a radio group. Because this process will (as suggested) remove the item from your log forever, an option is added to make a backup of the log file first. Later versions of MCLogger added checkboxes for when there is a not-so-useful string; “*Don’t append, just remove from log*” and, if there is a desire to send an item to HotString Helper for experimentation, but for some reason you don’t want to cull it from the log, use “*Don’t cull, just append*.” It is noteworthy that the button on the bottom always says “Cull and Append” even if you are not doing both.



As discussed below, several systems are in place to avoid logging un-useful strings. Despite that, I still log many manual correction items. Of course this will be different for other users, depending on their typing patterns. By the time my manual corrections log (MCLog.txt) was only three months old, it already had 2,200+ logged potential hotstrings, presently, about 5 months later, is more like 4,400+. And.... Many have already been removed, because the highest frequency reoccurring items have all been converted to new autocorrect entries in my *HotstringLib.ahk* file, and culled from the log. Because of the high number of items, running a frequency report currently takes me about six seconds. And... Since I keep removing the high-frequency items, what is left are a bunch of old single-occurrence “*singleton*” strings. Of course, any of the strings are singletons *at some point*—So it doesn’t make sense to remove *every* singleton. But it does make sense to bulk remove *really obscure* ones that will never have more than one occurrence. But how can we bulk-remove really obscure ones? Answer: I don’t know, so I made a function to remove all singleton items that are older than X number of days. How many days-old is too old? Define it in the variable that looks like this

```
AgeOfOldSingles := 180
```

Another Gui element is the context menu that appears when right-clicking the Windows System Tray icon .

It is recommended to put a link to the script in the Windows Startup folder and let it run in the background. There is an option for this in the SysTray Menu as well.

How it works

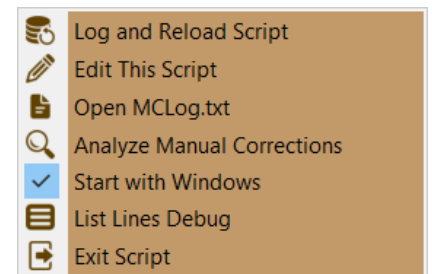
The MCLogger will watch for typing patterns that indicate you’ve typed a word and manually corrected the word. With each relevant keypress, it will analyze the cached characters and attempt to determine if a trigger-replacement pair can be garnered from the characters. For the purposes of identifying possible autocorrect items, we don’t care about digits or other characters, so those are ignored. Similarly, if the user uses the arrow keys to move back in a word, or clicks inside of the word, then the chain of keyboard presses will have been broken. As such, these actions will reset the “typoCache”. The Cache looks for letters, the BackSpace key, and the Space Key. To catch typos at the end of a sentence, periods are also grouped with letters. The watched-for pattern is: *At least three letters, followed by at least one backspace, followed by at least one letter, followed by a space*. This typoCache resets to blank if/when: a match is detected, the user clicks the left mouse button, or they press the Esc, Left, Right, Up, or Down key.

Early versions of the tool collected a lot of useless information and a lot of duplicates, so I added a filtering process. Trigger-replacement pairs are only kept if all of the following five conditions are met: (1) The replacement string is a match to a word from the word list file that is assigned near the top of the code, (2) the trigger string is NOT a word from the list, and (3) the current captured trigger-replacement pair is not the same as the previous one—to avoid duplicates. Condition (4) was added after I realized that, existing autocorrect items can still be logged. Consider this example: If I have an autocorrect entry

```
.*:creasion::creation
```

If I type the entire trigger string, then the word will be autocorrected, and I won’t manually correct it. But it is possible that I’ll type part of the trigger, then realize my error, such that my keypresses are this

```
creasio{Backspace}{Backspace}{Backspace}tion{Space}
```



This would not trigger the autocorrection item, but it would match the pattern the MCLogger watches for. I might then, inadvertently add the new hotstring “::creasion::creation” to my library, resulting in duplicate or conflicting items. To minimize the likelihood of this, I added condition four: If the trigger for the proposed to-be-logged hotstring conflicts with an existing autocorrect entry, don’t log it. Condition (5) was added more recently. To prevent “bad” hotstrings from getting added to my library, then removed for being bad strings, then inadvertently added again, a list of removed hotstrings (see next section) is now kept. Condition five is that the to-be-logged trigger item must not exist in the removed items list.

Trigger-replacement pairs are saved up in memory, then every X minutes (10 by default), they are logged to the MCLog.txt file. The “save intervals” stop if nothing is in the “saved up text,” and start again when there is. There is an *OnExit* command that will also append any saved-up text if/when the script is exited or restarted in the middle of an interval.

Pressing **Alt+Q** will provide a popup tooltip with the current cache and saved-up text, as seen in the screenshot to the right. Pressing **Shift+Ctrl+Win+Q** (or selecting the menu item, or selecting the button in hh2’s secret control panel) will analyze the data, checking for the frequency of occurrence for the items and sorting by frequency. The report will (optionally) be saved to the Windows Clipboard, and will appear in the form seen above, with the radio list. Like with DateTool-H, the nice big tooltips are made possible by Just Me’s [ToolTipOptions Class](#). With the two scripts (DateTool-H and MCL), the ToolTipOptions Class code is embedded in the script, so there is no need for an **#include** directive in the code.

```
Current typoCache:
s<asdf<asdfg<<sdf
=====
Current Saved up Text:
2024-04-16 -- ::frin::find
2024-04-16 -- ::digigs::digits
2024-04-16 -- ::backspcae::backspace
2024-04-16 -- ::thius::this
2024-04-16 -- ::reih::right
2024-04-16 -- ::pariicular::particular
2024-04-16 -- ::collected::collected
2024-04-16 -- ::condictios::conditions
2024-04-16 -- ::chahe::caches
2024-04-16 -- ::stcipt::stript
2024-04-16 -- ::strtarded::restarted
```

Removed HotStrings List

As indicated in the AutoCorrection- and Manual Correction-Log sections above, added in October of 2024, is the RemovedHotstrings.txt file. The entries are automatically generated when a hotstring is culled from the AutoCorrect log. They are formatted as:

```
Removed 10-30-2024 -> :*:alse::else
Removed 10-30-2024 -> :?:ualy::ually
Removed 10-30-2024 -> :*?:yuo::you (converted to word beginning and end items)
```

The list of previously removed items is read/accessed by the [MCLogger tool](#) and by the [Validation function](#) of HotStringHelper2. This is meant to reduce the likelihood of [reoccurring problematic hotstrings](#) in the HotstringLib.ahk file.

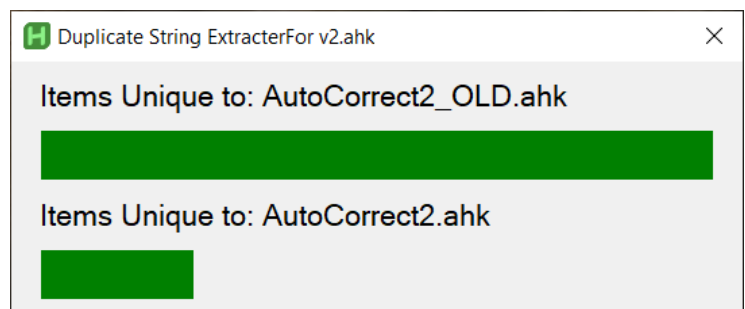
AutoCorrect Script Updater Tool

Despite the name, this tool doesn’t actually update your AutoCorrect Script. However, when you are manually updating your script, this tool might be useful.

From a [recent forum post](#):

It occurred to me that if people are using older versions of ac2, they might have added lots of cool new custom autocorrect entries via the HotString Helper tool. If those new custom hotstrings are still at the bottom of your ac2 file, then no problem... Copy-n-paste the new items over to the newer versions of ac2 as they are released... However... Some folks might like to alpha-sort their hotstrings. (At least that's what I do--LOL) If that is the case, how can you "bring over" your new custom items to the latest release of the ac2 script?

That's what this first script is for...



IMPORTANT:

- I left the file path to my own ac2 file as an example, but delete that, and use your own scripts and related directory/paths.
- Remember to scan the .ahk files, not the same-named .exe files.
- If you kept the name "AutoCorrect2.ahk", then you might rename that to something like, "AutoCorrect2_OLD.ahk"
- Put the newer version and the older version in the same folder as this script, so that the folder contains the files:

Duplicate String Extractor forV2.ahk

AutoCorrect2.ahk

AutoCorrect2_OLD.ahk

- If they are all in the same folder, you can probably just have the file names in the code, and not the whole directory path.
- If the Unique String Finder works correctly, It will create a text file with a list of the unique items, then it will open that text file and beep twice.
- Please note that this will not identify any unique Hotkeys or code that you added... Only hotstrings.
- Important: Hotstrings that are multi-line Continuation Section style hotstrings won't get extracted correctly either.

Conflicting or Nullified HotStrings Finder Tool

This script is also known as the *Dead String Locator*. While the above, *Duplicate String Extractor forV2.ahk* compares two scripts, identifying hotstrings that are unique to only the one assigned to the **extraFile** variable, *this one*, however, does an intra-analysis of a single script. It takes a lot longer to run though. The code used is the same as that of the Validation Checker of hh2, [discussed above](#). Each hotstring is successively compared against each item in the list, so 5k autocorrect items requires about 25 million passes. With each pass, five different conflict scenarios are checked for. My current autocorrect script has about 5400 items, and it takes just under 5 minutes.

Note that the script can't identify "triplicates." If there are three of the same item, it will flag them as two duplicate pairs. Also, with each pass (loop, actually), if a duplicate or conflict is found, the script immediately goes to the next item. This makes the script faster, but it means that, if a hotstring conflicts with other items in multiple different ways, the script might only identify the first problem. Running the script a second time might be necessary to find any additional conflicts.

Depending on your goal, you'll probably want to NOT scan the [Nullification Hotstring](#) items at the top, or the [Accented Words](#) at the bottom. As such, you can set the two variables near the top of the code.

ACitemsStartAt := 1980 ; <--- "AutoCorrect Items Start At this line number."

ACitemsEndAt := 6955 ; <--- stop comparing after this line number. No point scanning the non-English accented words.

Also, be sure to ensure that the **targetFile** variable is assigned the correct path.

When the script has completed, it will beep twice, write the report to a text file (saved in the same folder as the script) and then open that text file.

Color Theme Integrator Tool

From the code comments in the **ColorThemeInt** script: The WhiteColorBlackGradient function is based on [ColorGradient\(\)](#) by *Lateralus138* and *Teadrinker*. Calling the Windows color picker is also based on Teadrinker code.

I Used *Claude.ai* for debugging several parts and doing the "split complementary" math. Some terminology is from <https://colordesigner.io/color-wheel>

It is a Color Theme "**Integrator**" because it writes the colors to an ini file, which several of my other AHK gui-based tools (including HotString Helper2) read from.

The top "Reference" combobox reads from the colorArray, which has 120 elements, equidistantly circling the color wheel. I've attempted to create two color wheel options:

- * RGB uses "additive/light-based" color gradients.
- * CYM is uses "subtractive/pigment-based" color gradients.

Switch between these with the top radio buttons. Both arrays start at red, then loop around, back to red.

Using terminology from the above colordesigner site...

Given a reference color, its *Complementary* color will be directly across from it, on the opposite side of the color wheel. If, instead of choosing the Complementary color, you choose the two colors that are on equal-and-opposite sides of the Complementary color, then the three points will comprise a *Split Complementary* color set. The below tool uses the color selected in the combobox as the reference color and determines which color in the colorArray is its Complementary color. The first up/down box in the gui (*Split Size*) defines the number of steps from the Complementary position, that the other two colors should be.

The main three theme color variables are used:

- * **fontColor** is the color of the text on the gui and in applicable controls. I.e. `myGui.SetFont("c" fontColor)`
- * **listColor** is the background color of the listView, and other applicable controls. I.e. *BackgroundColor* as appears in the gui control options.
- * **formColor** is the GUI background color. I.e. `myGui.BackColor := formColor`

The fourth color is "outside of" the theme:

- * **warnColor** is the color used in place of the formColor, when the gui/app is in an alternate state.

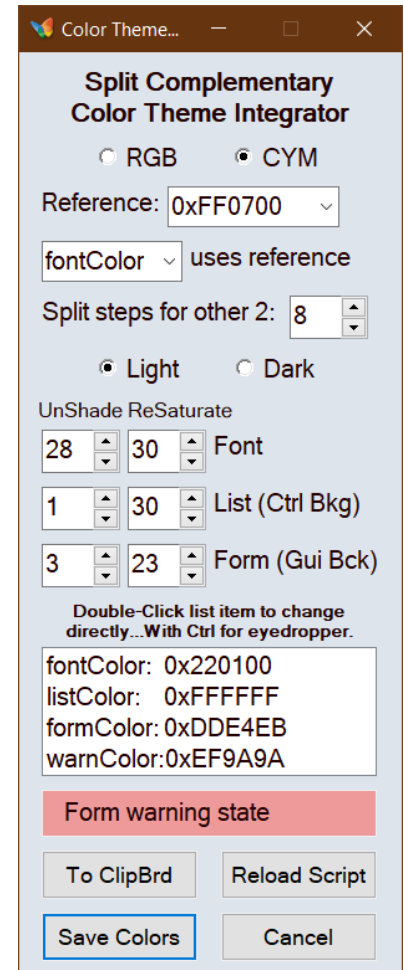
The colors are assigned by default as:

- * *fontColor* = The reference color that appears in the comboBox.
- * *formColor* = The color that appears in the posing Counter-Clockwise from the Complementary position.
- * *listColor* = The color that appears in the posing Clockwise from the Complementary position.

This means that the color which is chosen in the top comboBox corresponds to the font color of the form. The gui element (font/list/form) which is used for the reference color (and which corresponds to the top comboBox) can be changed with the second comboBox "... Is Reference."

The fourth color, *warnColor*, is not part of this color pattern scheme and is only changed via double-clicking the warnColor item on the bottom of the ListBox. However, when changing the light/darkness of the *formColor*, the *warnColor*'s level of light/darkness will change with it. If you don't have any projects with guis that use alternate Gui Backcolors, then you can ignore the warnColor.

So... If the *Split Size* is 10 to 15 or so, then the pattern will be *Split Complementary*. If the Split Size is set to 0, then the listColor and the fontColor will be at the same position and will be *Complementary* to the formColor. If the Split Size is the max of 60, then all three colors will be at the same location and the theme will be *Monochromatic*. If the Split Size value is around 45 to 50, then the



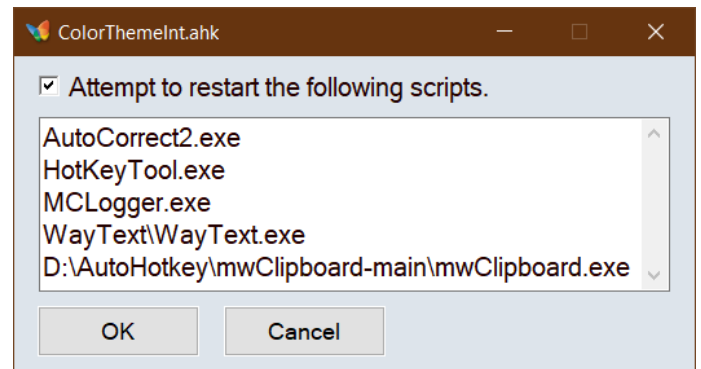
pattern will be “Analogous” (again, see colordesigner site). And if the Split Size is 20, then the pattern will be a “Triad.” As indicated above, the primary color is set in the top comboBox, and the other two are the split colors. The split colors can swapped by setting a negative number as the Split Size.

It is noteworthy that, with a gui, the font needs to “stand out” from the background color of the gui and the controls, so the colors chosen will often need to be adjusted in terms of *lightness/darkness*. Typically the font will be on one end of the light/dark continuum, and the background colors will be at the other end. The Light/Dark radios swap this. The bottom three up/down boxes are for fine-tuning the light/darkness. The warnColor must be manually changed, but it will change light/darkness as the formColor does.

Similar to changing the shading, users may wish to “tone-down” one or more of the colors by reducing the *saturation*. This is simulated by fading the color to gray. The saturation up/down controls are next to the shading ones. Double-clicking the **ReSaturate text label** will reset the saturation levels to max. And double-clicking the **UnShade text label** will set the shade spinners to the middle number (neither light, nor dark.)

Near the bottom is the 4-row ListBox. This shows the hexadecimal color values that will get exported. The ListBox is also an “override.” If you can’t get a color you like with the *Hue, Shade, and Saturation* controls, double-click the row in the list to pick a totally different color. This will call a Windows color picker (*based strongly on Tadrinker code*) and disregard the above gui controls. Holding the **Ctrl Key** while double-clicking will allow you to “eyedropper-pick” a color from the screen. Note that if you then change the Hue, Shade, or Saturation controls, those controls will, again, take precedence and determine the colors used for the first three items (font, list, form), thus you will lose your custom-picked color. A warning will appear before this happens.

The current four colors, as well as the values of all the controls, can be saved to a configuration (*ColorThemeSettings.ini*) file. The tool will attempt to read from the file at start. If the ini file is not found when the script starts, default values will be applied. Near the top of the code is the “**restartTheseScripts**” list. The tool will look for a list of scripts to restart (so that their gui colors will be updated.) When the save button is pressed, the user will be given the option to also restart these scripts. Uncheck the checkbox to bypass restarting the other scripts. (See also screenshot on right.) Please note that the ColorThemeInt tool won’t intelligently find the scripts that need to be restarted—that list must be manually added to the *restartTheseScripts* variable in the code.



HotString Helper2 will attempt to check if the *ColorThemeSettings.ini* file exists, and if it does, will add an additional button to the bottom of the *Secret Control Panel*. The button starts the *ColorThemeInt* application and launches the main form directly. This allows users to change the color of the hh2 form. The script looks for command line arguments when it starts. If there are command line arguments, the gui is shown at startup, and Esc or closing the form exits the app. this was added so that the tool could be launched from the HotString Helper2 tool.

End

Please post a reply on the [AutoCorrect for v2 forum thread](#), or on the [GitHub/AutoCorrect2](#) page, if you find any errors, or have any suggestions, for the above discussed code, or for this manual.

Thanks for reading, and a big THANK YOU to the people who have helped me create this.

-kunkel321

